WEIGHTING SCHEME FOR THE SPACE SURVEILLANCE NETWORK AUTOMATED TASKER

THESIS

Beth L. Petrick
Captain, USAF

AFIT/GSO/ENS/94D-13

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GSO/ENS/94D-13

WEIGHTING SCHEME FOR THE SPACE SURVEILLANCE NETWORK AUTOMATED TASKER

THESIS
Beth L. Petrick
Captain, USAF

AFIT/GSO/ENS/94D-13

Approved for public release; distribution unlimited

AFIT/GSO/ENS/94D-13

WEIGHTING SCHEME FOR THE SPACE SURVEILLANCE NETWORK AUTOMATED TASKER

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Space Operations

Beth L. Petrick, B.S.

Captain, USAF

DECEMBER, 1994

THESIS APPROVAL

STUDENT: Beth L. Petrick, Capt, USAF

CLASS: GSO-94D

THESIS TITLE: WEIGHTING SCHEME FOR THE SPACE SURVEILLANCE NETWORK AUTOMATED TASKER

DEFENSE DATE: 22 November 1994

COMMITTEE:

Name/Title/Department	Signature
Advisor: Roger C. Burk, Maj, USAF, PhD Assistant Professor of Operations Research Department of Operational Sciences Graduate School of Engineering	Roger C Burk
Reader: James T. Moore, Lt Col, USAF, PhD Associate Professor of Operations Research Department of Operational Sciences Graduate School of Engineering	James T. Moore
Reader: James S. Shedden, Lt Col, USAF, PhD Assistant Professor of Operations Research Department of Operational Sciences Graduate School of Engineering	James S Shedden

ACKNOWLEDGMENTS

I would like to thank my sponsor, the members of the First Command and Control Squadron. In particular, I would like to say a special thanks to Capt Danny Combs and Warrant Officer Allan Brenton. They helped me define my problem and answered a myriad of questions concerning the operation of the Prototype Tasker. I also extend my deepest gratitude to Gil Miller and Mark Brocco of The MITRE Corporation. They adapted the Prototype Tasker to my research approach and also passed on valuable information.

Dr. Lester Ingber developed the ASA algorithm used in the research. I would like to thank him for letting me use the software and responding quickly to my questions.

I would also like to thank the members of the National Aerospace Intelligence Center for providing computer facilities and support. The members of the SCDE office, especially Rita Peters, were very helpful. Without their invaluable assistance, I would probably still be trying to get the software to work.

My committee was very influential throughout the course of the research. I would like to thank my advisor, Maj Roger Burk, for giving me guidance when I needed it, but also letting me do the research on my own. My readers, Lt Col James Moore and Lt Col Jim Shedden, gave me more assistance than just reading the drafts. They were involved in finding an approach to my problem and have answered many questions in the past seven months.

Last, but not least, I would like to thank my parents and my fiancé for listening to my worries and fears during the past months. Mom, thanks for your words of wisdom and helping me keep things in proper perspective.

Beth L. Petrick

Table of Contents

Page

Page

List of Figures

List of Tables

ABSTRACT

The purpose of this research is to find the best weighting scheme in the SPADOC 4C Sensor Tasking Prototype. This software, known as the Prototype Tasker, assigns a tasking to each sensor of the Space Surveillance Network. A tasking is a list of objects in which the sensor needs to gather positional data, called observations. One of the inputs to the Prototype Tasker is a user-definable weighting scheme. The goal is to find the weighting scheme that produces the most efficient taskings.

This problem was solved using a Simulated Annealing algorithm. The values in the weighting scheme were the variables of the problem. The objective function was a measure of the goodness of each tasking produced. The Simulated Annealing algorithm was set to vary the weighting scheme and find the one that produces the tasking with the highest objective function.

There were four successful executions of the simulated annealing algorithm. Each one produced a different weighting scheme. However, there were noticeable trends in the relative magnitudes of the weights. Also, it was noticed that a slight decrease in the number of observations on the taskings will increase the expected amount of information gathered by the Space Surveillance Network.

WEIGHTING SCHEME FOR THE SPACE SURVEILLANCE NETWORK

AUTOMATED TASKER

I. Introduction

1.1 Background

1.1.1 The Space Surveillance Network. Space surveillance is described as "detecting objects as they enter space, detecting events caused by objects in space as they occur, and confirming an object has departed space" (18:2). It is one of the missions performed by the United States Space Command (USSPACECOM). Maintenance of a satellite catalog is one of the important tasks of space surveillance.

Since the early years of the space era, the United States has cataloged the manmade satellites and debris circling the earth. This satellite catalog contains the orbital element sets for those space objects large enough to detect. This orbital information is used to chart the current position of the objects and predict their future orbits. It is used to predict collisions and close approaches between satellites and orbiting debris, and also aids in predicting the re-entry of these objects into the earth's atmosphere (18:2).

The satellite catalog has been kept since 1957 and over 21,000 objects have been cataloged since then (12:98). This number includes those objects that have since de-orbited or broken into smaller, undetectable pieces. The catalog currently contains about 8000 objects (8). The objects in the catalog include active and inactive satellites, rocket bodies, and pieces of debris.

In order to keep the satellite catalog up to date, precise data are gathered on the position of the objects. This information is gathered by the Space Surveillance Network (SSN), which consists of twenty-two sensors placed world-wide. These sensors include radars, electro-optical sensors, and passive radio frequency sensors (26:84). The sensors detect space objects and gather positional data on them. One single point of positional data is called an observation. Currently, there are over 40,000 observations performed by the SSN per day (11:569; 18:4). These observations are then sent to the Space Control Center (SCC), formerly the Space Surveillance Center, located at Cheyenne Mountain Air Station (CMAS) in Colorado.

The SCC is responsible for the command and control of the SSN. Its mission includes satellite catalog maintenance, object identification and the monitoring of special events (12:97). The SCC is part of the Space Defense Operations Center (SPADOC), which is responsible for surveillance, protection and negation of space systems (26:91). SPADOC is under the direct jurisdiction of USSPACECOM. The 1st Command and Control Squadron (1 CACS) is a squadron within the Air Force component of USSPACECOM, Air Force Space Command (AFSPC). As part of the SCC, the 1 CACS performs the routine operations of the SCC. Among these everyday operations is the tasking of the SSN. It performs this using the SPADOC 4C Sensor Tasking Prototype, commonly known as the Prototype Tasker.

1.1.2 Prototype Tasker Overview. Currently, the 1 CACS is using the Prototype Tasker to perform the daily tasking of the SSN. The Prototype Tasker is a predecessor to the tasker that will be in the SPADOC 4C software upgrade. The prototype, developed by The MITRE Corporation, was built to verify the logic and feasibility of the SPADOC 4C Tasker (5:94). It has been used operationally since mid-1993. The Prototype Tasker is explained in detail in Chapter 2. A brief overview is given below.

The Prototype Tasker gives a daily tasking to each of the sensors. The tasking is basically a list of all of the objects for which a sensor needs to obtain positional data. Associated with each object is the number of tracks that sensor needs to perform for that object on that day. A track is the collection of observations obtained during one pass of an object. A pass is one period of time in which the space object is visible to the ground sensor. In other words, a pass is the time between when the object rises above the horizon of the sensor and then sets (8). A track, therefore, is all of the observations made on one object within one of its passes over a sensor. The Prototype Tasker tasks tracks, not observations, when performing its function.

The logic of the Prototype Tasker can be broken into four major parts. First, the list of all objects is prioritized. Second, for each object, it determines the number of tracks and number of different sensors that are needed to properly update the object's orbital element set. Third, it determines which sensors will be allocated tracks for the object. Fourth, it compiles a daily tasking message. This is done after all objects have been considered for tasking. The daily tasking message is sent to each sensor. From the message, each sensor learns which objects it needs to track that day and the number of tracks it needs to perform on each object.

The sensor allocation portion of the Prototype Tasker uses a greedy algorithm. The Prototype Tasker works down the list of objects in descending order of priority. For each object, it determines if the prior tasking is acceptable. If it is, this tasking is used. If not, it calculates visibility of the object at each site and then ranks the sensors from highest to lowest. This ranking is done for each sensor that has visibility on that particular object and is available for tasking.

In order to do the ranking of sensors for an object, the Prototype Tasker computes a score for each of seven criteria. The computation of these scores uses a set of numbers contained in a user definable file. This weight file is commonly known as the "weights" or "weighting scheme." The user, 1 CACS, has the task of choosing these weights. The Prototype Tasker then uses the weights in determining a score for each of the seven criteria. The seven scores are added, and this sum gives the total score, or "goodness," for a sensor for that particular object. The sensors with the highest total scores are tasked to get some of the tracks of that object for that day.

1.2 Conduct of the Research

1.2.1 Problem Statement. There is no current mathematical insight into what the user-definable weights should be. The 1 CACS has requested research to determine what the weights should be to get a good or optimal overall tasking.

1.2.2 Procedure. This problem is modeled as a numerical optimization problem. The decision variables, or domain, are the weights. Any other information the Prototype Tasker uses as input is assumed to be unchangeable parameters of the problem. This includes the lists of objects and sensors, and any information contained in the database of the Tasker. The Prototype Tasker performs a function on the parameters and the variables. The end result, or range of the problem, is the daily tasking. Once the daily tasking is produced, the "goodness" of the tasking needs to be found. One way of determining this "goodness" or merit is to use statistics from the tasking itself. When the Prototype Tasker performs a tasking, it keeps track of how many tracks were required and how many were actually included in the tasking. It also keeps track of the probability of acquisition of the tracks on the tasking. The probability of acquisition is the probability that the track will be performed by the sensor on an object, given that the sensor is tasked with a track of that object. The Prototype Tasker also evaluates other statistics, but the ones mentioned above were used in evaluating the merit of the tasking. This merit will be the objective function of the problem. The goal of the research is to find the values in the weight file that will produce the tasking with the maximum possible merit.

An iterative search was used to find the appropriate weights. Basically, different weighting schemes were put into the Prototype Tasker, while holding all of the other data constant. At each iteration, the values in the weight file were changed. The Prototype Tasker was run with the new values that produced a new tasking. The merit of that tasking was evaluated and checked against the previously found merit values. However, this problem is full of difficulties. The numbers in the weight file are integers, causing a discrete domain. In addition, the objective function is discrete: A small change in the weighting scheme may not change the taskings produced by the Prototype Tasker. However, if the changes in the weighting scheme are larger, then the tasking may change. Another difficulty is the lack of an analytical objective, or merit, function. All of these complications make it difficult to apply classical non-linear optimization methods. In order to employ the classical non-linear optimization methods, one has to assume that the function that you are trying to optimize, f(x), is continuous (24:75).

One type of non-linear optimization method is the direct iterative search. An iterative search method starts at an initialization point. The search method evaluates f(x) at the current point. It then chooses a point in the neighborhood of the initialization point, and evaluates f(x) at it. If the value of f(x) at the new point is better than at the current point, then it moves to that new point. The new point becomes the current point and the search continues again from this position. The process continues until the changes in values of f(x) are smaller than some previously chosen value.

Direct search methods assume a continuous f(x) and can only guarentee finding local optimums. The function performed by the Prototype Tasker is not continuous, and a global solution is preferred. With this in mind, we decided to attempt to solve the problem by using a Simulated Annealing (SA) algorithm. SA is an iterative search algorithm that does not have difficulties with discrete objective functions. In many cases, it also is able to find global optimal solutions as opposed to local optimum (16:27).

1.2.3 Challenges. The purpose of the research is to find the best weighting scheme for the Prototype Tasker. The approach taken was to model this problem as a numerical optimization problem. A Simulated Annealing algorithm was used to complete the research. Though this may seem to be not very difficult to accomplish, there are many challenges to this particular problem. This section reviews some of the obstacles that needed to be overcome before the actual optimization could be performed.

First, this problem was ill defined. The sponsor requested research that would find a weighting scheme that provided the best utilization of the SSN. However, the formulation of the problem and the objective functions had to be defined by the researcher.

The second challenge was the Prototype Tasker itself. The databases for the Prototype Tasker required one Gigabyte of memory. It also had to be placed on a computer platform on which it had never been run. Another hurdle was the execution time of the Prototype Tasker. Each execution of the Prototype Tasker takes over one hour to complete. Therefore, the execution time needed to be drastically decreased in order to apply an iterative search. Another complication resulted from the interactive nature of the Prototype Tasker. Doing an iterative search on an interactive program would be impossible. Therefore, before the research could begin, a new non-interactive version of the Prototype Tasker had to be provided. After the non-interactive version of the Prototype Tasker was in place, it then had to be integrated with the Simulated Annealing algorithm in order to accomplish the iterative search.

1.2.4 Limitations and Assumptions. Though research is ongoing in many different portions of the Prototype Tasker, this thesis was limited to finding appropriate numbers for the weighting scheme. The research was held within the bounds of the current algorithms used in the logic of the Prototype Tasker. In other words, the Prototype Tasker was viewed as a "black box."

The largest limitation of this project was time. Due to the fact that each run of the Prototype Tasker takes over one hour to complete, we were limited in the number of times the Prototype Tasker could be run with a given set of parameters. This is especially difficult when using an iterative search.

Another limitation is in the choice of the objective function. In reality, we would like to see the overall effect of the tasking produced by the Prototype Tasker. This can be seen either as a response of the SSN to the tasking or as overall quality of the satellite catalog. However, neither of these options is feasible. This would require either testing different weighting schemes on the operational system or employing a simulator. On the operational system, each data point would take one day to complete. This is not feasible within the time limits placed on the research. Also, experimental research on an operational system could lead to real world problems. Another alternative would be to simulate the response of the SSN and the updates made to the satellite catalog. However, no such simulator currently exists. Building a simulator would be extremely complex and is beyond the scope of this research. For these reasons, objective functions had to be developed to estimate the performance of a tasking in the SSN.

The purpose of the research was to find a weighting scheme that would produce taskings with improved utilization of SSN resources. Therefore, it is assumed that some weighting scheme will produce the best possible taskings and that SA will be able to find this weighting scheme.

II. The Prototype Tasker

2.1 Introduction

Chapter 1 gave a summary of the Prototype Tasker. Here, the logic used by this software is described in more detail.

There were several goals for this piece of software (8). First, it was to decrease the amount of manual interactions needed in the daily tasking process. Second, it was to improve on the quality of the element sets in the satellite catalog. Lastly, it was to help in making the utilization of the SSN more efficient.

The Prototype Tasker tells the sensors on which objects to gather observations. It does not, however, produce a schedule. It is up to each individual sensor to schedule the times for each observation it will perform during the day. Each sensor has its own separate requirements and missions. Therefore, it would be very difficult to make a time schedule on a global basis for the SSN.

Though the Prototype Tasker has been used with excellent results, it is believed that the algorithm can be improved. Presently the Prototype Tasker uses a "greedy" algorithm to perform its functions. However, if a strict optimization algorithm was used, the quality of the taskings produced would improve. The basic purpose of the Prototype Tasker is to assign tracks to the sensors of the SSN. This can be modeled as an assignment problem. However, this aspect of the problem was not addressed in this research. Due to the sponsor's requirements, the Prototype Tasker was treated as a "black box" and its algorithms were not changed.

2.2 Tasking Groups

Before going into the logic of the Prototype Tasker, tasking groups need to be explained. Tasking groups are a key concept in SSN tasking and in the Prototype Tasker. Basically, a tasking group is a group of objects that have similar tasking characteristics

(21). The main distinguishing features across the tasking groups is object type and orbit class.

The tasking group number consists of three digits. The first digit represents the object type. The object type numbers and their description are given in Table 2.1. For object types 1xx - 5xx, the last two digits represent the orbit class. These last two digits run from 01 to 65. The orbit class is a description of the shape of the orbit of the object. It is given in ranges of altitude of the apogee, the point in the orbit farthest from the earth, and perigee, the point in the orbit closest to the earth. Examples of the first four orbit classes are given in Table 2.2. About 95 percent of the objects are contained within the

Table 2.1 Object Types (14:1)

Number	Description
0	Analyst's Use
1	Small Debris
2	Large Debris
3	Rocket Body
4	Inactive payloads
5	Low Interest Active Payloads
6	High Interest Payloads
7	Decaying Satellites
8	Analyst's Use
9	Analyst's Use

Table 2.2 Examples of Orbit Classes (7)

	Apogee Altitude		Perigee Altitude	
Orbit Class	Min (km)	Max (km)	Min (km)	Max (km)
1	0	300	0	300
2	300	575	0	300
3	300	575	300	575
4	575	1000	0	300

1xx - 5xx tasking groups (7). For the other object classes, the last two numbers represent either the object's orbit or mission.

The tasking groups hold important tasking information needed by the Prototype Tasker. Each object in the catalog belongs to one of the several hundred tasking groups. When the Prototype Tasker is determining a new tasking for an object, it uses the information contained in the definition of the object's tasking group. This information includes the tasking interval, the sensor ranking list, the sensor selected preferences and the tasking table (21). These items will be discussed within discussion of the Prototype Tasker's algorithm.

2.3 The Algorithm

Each day, before the actual tasking is done, there are a couple of preliminaries that need to be performed (8). First, the observations from the previous day are imported into the database. Second, an Element Quality Prediction (EQP) is performed. This step uses historical data to measure how well past element sets predicted errors in observations (8; 5:95).

The flow chart in Figure 2.1 gives a basic outline of the procedures in the Prototype Tasker. A brief description of the chart is given in this paragraph. The details of each item in Figure 2.1 are given in full detail later in this chapter. The first step is prioritizing the satellite list. Next, the Prototype Tasker deletes the sensors that are not available to perform tracks for that day. The logic then enters the satellite loop. The items in the loop are processed for each object in the satellite list. Once in the loop, the Tasker processes information brought in from the SSN. It then evaluates the tasking that the object has from the previous day. If that tasking is acceptable, it is used. If not, it reevaluates the number of observations needed for that object for the day. It then takes that information and assigns the tasking. This is where the values in the weight file are applied. After a tasking is assigned to the object, whether it be a new tasking or the one

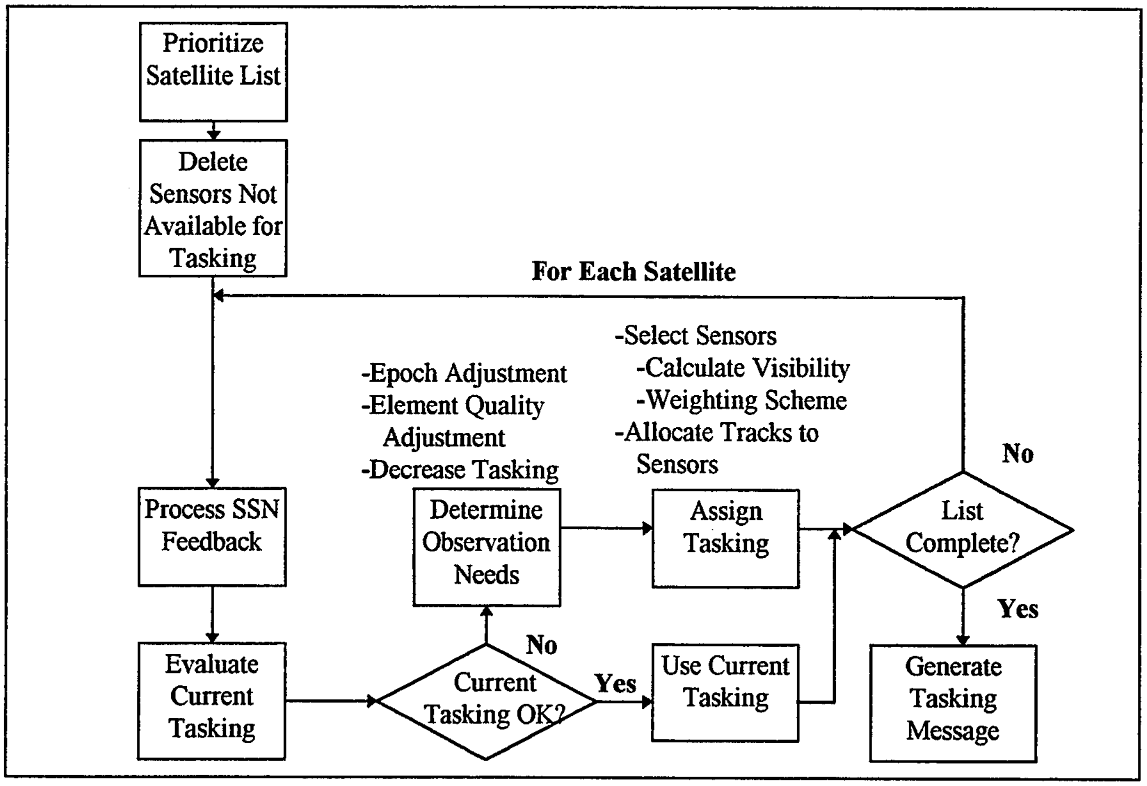


Figure 2.1 Prototype Tasker Design (21)

from the previous day, the Prototype Tasker moves to the next object in the satellite list. This is done until all objects have gone through the loop. After that, the tasking message is generated.

2.3.1 Prioritizing the Satellite List. The objects in the satellite list are prioritized according to five different criteria (21). These are discussed below in their order of importance. The objects are grouped according to the first criterion. These groups are then further divided by the second criterion. This division continues until all five criteria have been considered.

First, the objects are broken into categories. A category number is attached to each of the objects in the satellite list, and the number ranges from one to five, one being the highest priority. The category number is computed in the "Determine Observation Needs" portion of the Prototype Tasker. The category calculated for the object in the previous tasking is the category used here. The categories inform the sensors which objects take precedence in case of a schedule conflict at the site (5:94). When setting up the ordered list, all of the category 1 objects will be tasked before all of the category 2 objects, and so on down to category 5.

The next criterion is either manual or automatic tasking. Those objects that require manual tasking take priority over the rest of the objects in the same category. This procedure is not available in the Prototype Tasker but will be used in the SPADOC 4C Tasker.

Next, the objects are sorted according to the period of their orbit. They are placed into one of four classifications. Those objects in a geosynchronous orbit are tasked first. Semi-synchronous objects follow them, followed by deep-space objects, and finally near-earth objects. Objects at higher altitudes will not be visible to as many sensors as those objects at lower altitudes. Therefore, the priority is given to those objects at higher altitudes (5:95).

After the objects are grouped according to the previous three criteria, they are then ranked according to the inclination of the orbits. Objects with lower inclination are given precedence. Objects of low inclination are not visible to as many sensors as those with higher inclinations. Since the number of sensors that can track low inclination objects is more limited, the low inclination objects are given a choice of sensors before the rest (9).

The last tie-breaker is the satellite numbers. Each object is given a sequential identification number once it enters the satellite catalog. Hence, the more recent objects have higher satellite numbers. The priority goes to the objects with higher numbers since newer objects are more likely to have problems than older objects (8).

2.3.2 Evaluate Current Tasking. Once the prioritization of the satellite list is complete, sensors that are out of commission for the day are deleted from the list of available sensors. The algorithm then enters the satellite loop. One iteration of the loop is performed for each object in the satellite list. The algorithm works down the list in descending order of priority. It stops when all objects have been considered for tasking.

The first step in the loop is the processing of historical data and feedback. These computations are used in the rest of the steps in the satellite loop. They are described in the step in which they are used.

The next step is to evaluate the current tasking. Basically, the Prototype Tasker looks at the tasking the object had in the previous tasking. It then decides if this tasking will be sufficient for the new tasking. If the previous tasking is acceptable, then the object keeps the same tasking. In this case, the object is done, and the loop moves onto the next object. Research done by the 1 CACS and The MITRE Corporation has shown that, on average, 80 percent of the objects use their previous tasking during each daily run of the Prototype Tasker (22).

There are quite a few ways that the current tasking can fail (22). Several of them are mentioned here. First, if one or more of the sensors assigned for this object is unavailable for tasking, then the object's tasking needs to be evaluated again. This can happen if there is an outage at the sensor or if the sensor is already at its maximum capacity. Another way is if the visibility of one or more of the sensors cannot support tracks on that object. Two other ways are through tasking adjustments and passing a tasking interval.

There are two types of tasking adjustments, epoch and quality (8). An epoch adjustment is made when the epoch age of the object is larger than the epoch threshold. The epoch age is defined as the current time minus the epoch. The epoch for an object is the last ascending node prior to the last observation in the software. The epoch age, therefore, roughly gives the time since the last observation. The epoch threshold is defined for the object's tasking group. The Prototype Tasker uses this threshold number. If the epoch age is past this threshold, a new tasking is generated. The other type of tasking adjustment is in element quality. This is associated with the element quality prediction mentioned in the beginning of Section 2.3. Element quality is measured in kilometers per day. It represents the rate at which the errors in the element set are increasing (8; 5:95). The element quality is compared to the element quality threshold. This is done in the same manner as the epoch adjustment in that the threshold is defined within the tasking groups. When the element quality goes beyond the threshold, the object is given a new tasking.

Each tasking group also has a defined tasking interval. This interval tells how often each object in the tasking group needs to have its tasking reevaluated (8). When the current tasking is checked, it also checks the tasking interval. When the tasking interval is meet, the object's tasking is evaluated.

2.3.3 Determine Observation Needs. If the current tasking is not acceptable, a new tasking is determined. The next step is to figure out how many tracks at how many different sensors are required for that day on that object.

The Prototype Tasker uses tracks as its basic element when assigning sensors to an object. A track is made up of observations made during one pass of visibility. The number of observations in a track is different depending on sensors themselves (8).

This phase of the tasking uses what is called a tasking table (Figure 2.2). There are several tasking tables defined within the Prototype Tasker. Each tasking group declares its appropriate tasking table. Therefore, more than one tasking group may have the same tasking table. The rows signify the category of the object and the columns show an increase in the amount of tracks or sensors required. "Category" in the table is the same category introduced in "Prioritizing the Satellite List." The categories of the objects inform the sensors which objects take precedence in case of a schedule conflict at the site.

Category	Number of Tracks/Number of Sensors							
1	1/1	2/1	3/2	4/3	5/3	6/4	7/5	8/6
2	1/1	2/1	3/2	4/3	5/3	6/4	7/5	8/6
3	1/1	2/1	3/2	4/3	5/3	6/4	7/5	8/6
4	1/1	2/1	3/2 Nominal	4/3	5/3	6/4	7/5	8/6
5	1/1	2/1	3/2	4/3	5/3	6/4	7/5	8/6

Figure 2.2 Example Tasking Table (21)

The shaded area in the table represents all of the possible configurations for objects in one particular tasking group. The nominal entry is in the lower left-hand corner of the shaded area. The Prototype Tasker uses these tables to determine the changes needed in tasking.

During any run of the Prototype Tasker, the object's tasking is represented by one of the shaded boxes in the tasking table. This current box is the one calculated during the previous tasking. If the epoch threshold was broken, the current tasking will move up one block within the shaded area. If that is not possible, it will move to the right one block. If an adjustment is needed because of element quality, then the current tasking will move one block to the right, and up if a move to the right is not possible. If both thresholds are broken, then the epoch adjustment takes precedence and a move is made upwards, if possible (8). During this procedure, the object's category may change. However, this will not change its current position in the satellite list. The new category will be used when prioritizing the list the next time the Prototype Tasker is run.

If the object has entered this step because of the tasking interval requirement, then it is possible for the tasking to decrease (8; 21). If there has been no epoch or element quality adjustments made during the last tasking interval, the object will move back one block in the tasking table. It attempts to decrease the category first. If it is on the lowest possible row, then a move is made to the left to decrease the tasking. If the object was at the nominal entry to begin with, it will stay there.

The end result of this is that the object has the number of tracks and sensors needed to properly update its element set. The current tasking, or box in the tasking table, is saved as the starting point for the next tasking that will be calculated. Determining the observation needs leads into the final step, assigning the tasking for that object. This is where sensors will be tasked to perform the tracks.

2.3.4 Assign Tasking. The final step is the determination of the sensors that will be tasked with tracks for that object. The first thing that is done is retrieving the sensor candidate list. This list comes from a sensor ranking list that is defined within the tasking group record (5:96). The ranking list includes all possible sensors that can make observations on the objects in that tasking group. The sensor ranking list includes a preference value of each of the sensors for the objects in that tasking group. However, these numbers are not considered at this point. The sensors in the ranking list become the initial sensor candidate list for the object. Those sensors that are not available for tasking or do not have adequate visibility are dropped from the list. If the number of sensors left on the updated list is less than or equal to the number of sensors needed that day for that object, those sensors are used. The algorithm then enters another loop. At each iteration of the loop, the best sensor is determined by the total weighting score, which is discussed later in this chapter. This sensor is selected and is removed from the candidate list. This loop continues until the required number of sensors is assigned to the object. The tracks are spread out as evenly as possible among the selected sensors (5:97). If there is an uneven distribution of tracks, the higher number goes to the first chosen sensor. For example, suppose our object needs four tracks at three sensors. The first sensor will receive two tracks. The second and third ranked sensors will each get one.

The sensors are ranked according to a weighting algorithm. Each available sensor receives a score for each of seven criteria. The total score is computed as follows:

$$\begin{aligned}\textit{sensor weight score} = {} & \textit{sensor selected weight} + \\ & \textit{rank weight} + \\ & \textit{\# of passes weight} + \\ & \textit{\# acceptable passes weight} + \\ & \textit{orbit distribution weight} + \\ & \textit{loading weight} + \\ & \textit{probability of acquisition weight}\end{aligned}$$

Each one of the seven individual weighting scores is evaluated within their own function in the program.

Before these weighting scores are computed, the user-definable weighting database is read. This weight file is called *weights.txt* and it is an ASCII file which the 1 CACS can alter. This weight file contains numbers that are used in the computation of the seven scores. Appendix A gives the current setting for the weight file. There are several numbers in the weight file that are not used in the Prototype Tasker. These include the Above/Below Line, Currently Tasked, and Sensor Preference weights. The Above/Below Line and the Currently Tasked weights are not used by the Prototype Tasker, but they will be used in the SPADOC 4C Tasker. The Sensor Preference weight is included in the code of the Prototype Tasker, but the definitions needed to use it have not yet been set in the tasking group records. All of the information on the computation of the weights has been gathered from the code of the Prototype Tasker (25) unless otherwise noted.

Sensor Selected Weight. This weight is computed from a flag set in the tasking group containing the object. Each candidate sensor is searched to see if it does have the sensor selected flag set to true for that tasking group. If it does, the sensor selected weight equals the *selected_weight* in the weight file. If not, the sensor selected weight equals the *not_selected_weight*, which is 0.

The purpose of this weight is to assure that a certain sensor will be assigned to an object. This is only used on tasking groups including about one half of one percent of the overall satellite catalog (10). Note that the *selected_weight* is currently 1000, which is at least an order of magnitude higher than the other numbers in the weight file.

Rank Weight. This weight is determined from the sensor ranking lists mentioned before. Each tasking group has a sensor ranking list associated with it. The ranking list contains all of the sensors that are able to track objects in that tasking group. With each sensor is a ranking from one to ten. If the sensor in question is scored as "1" in the ranking list, the rank weight would equal the first number under *rank_weights* in the weight file. For example, using the file in Appendix A, the *rank weight* will equal 20. A score of "2" in the ranking list will give a *rank weight* of 18.

Number of Passes Weight and Number of Acceptable Passes Weight. Both of these weights are essentially found in the same manner. First, the number of passes for the object over each sensor is found. This number includes those passes that occur during the day in which the tasking is being generated. The difference between the two weights is that the number of acceptable passes weight takes into account the elevation of the pass. If the elevation is lower than a minimum set for that sensor, then that individual pass is not considered. The number of passes weight does not make this check on the elevation angle.

Once the number of passes is found, both of these functions have the same logic. It first determines if the number of passes for the object on the sensor is at least the number of tracks that sensor would need to perform. The number of tracks is determined by dividing the total number of passes required by the number of sensors required. The "left over" tracks, if they exist, are given to the first tasked sensor. If the number of passes is at least the number of tracks, it then calculates an additional pass weight:

$$\textit{additional pass weight} = (\#\ \textit{of passes} - \#\ \textit{of tracks needed}) \times \textit{additional\_no\_passes\_weight}$$

The *additional_no_passes_weight* comes from the weight file. The additional pass weight score can be no larger than the *max_additional_pass_weight* in the weight file. The way it has been employed in the past, the *max_additional_pass_weight* is the same as the *additional_no_passes_weight*. The effect is that if you have one or more extra passes, the additional pass weight score equals the *additional_no_passes_weight*, which is also the *max_additional_pass_weight.*

If the number of passes is at least the number of tracks required, then the overall number of passes weight is calculated by adding the *required_no_passes_weight* from the weight file and the additional pass weight above. If there are not enough passes, the number of passes weight is solved as:

$$\textit{number of passes weight} = \textit{required\_no\_passes\_weight} \times \frac{\#\ \textit{of passes}}{\#\ \textit{of tracks needed}}$$

These formulas for the number of passes weight are the same for the number of acceptable passes weight.

Orbit Distribution Weight. One condition that enables better updates of the element sets is to gather observations on the object from different parts of its orbit. The orbit distribution weight takes this into consideration by giving higher scores for sensors that are more widely spaced across the globe. This weighting score is developed differently for different types of objects. The geostationary objects use a function of longitude of the sensors. All other objects use the latitude of the sensors. This weight is also determined differently depending on whether or not sensors have already been tasked to the object.

For the geostationary objects, the function uses the longitude of both the satellite and the sensor. If no sensors have been tasked for this object yet, the orbit distribution weight is given by:

$$\textit{orbit distro weight} = \left| \frac{\textit{sensor longitude - longitude of object}}{\textit{orbit\_distribution\_weight}} \right|$$

The *orbit_distribution_weight* is given in the weight file. If sensors have already been selected, a higher weight is given to those sensors farther away from those already tasked. The algorithm works through the current list of candidate sensors. For each one, it finds the tasked sensor that is closest in longitude to it. The orbit distribution weight is given as:

$$\text{orbit distro weight} = \left| \frac{\text{closest tasked sensor longitude} - \text{candidate sensor longitude}}{\text{orbit\_distribution\_weight}} \right|$$

For the rest of the objects, the calculation is very similar. If no sensors have yet been tasked with the object, the candidate sensors closest to the equator are given higher weights.

$$\text{orbit distro weight} = \left| \frac{90.0 - \text{candidate sensor latitude}}{\text{orbit\_distribution\_weight}} \right|$$

If sensors have already been tasked, the weights are calculated the exact same way as for the geostationary objects, except longitude is replaced with latitude.

Loading Weight. The loading weight is a function of how much of the sensor's capacity has been used thus far on previous objects. Those sensors with more capacity available will get a higher loading weight. Once the used capacity reaches above 105 percent, the weight is essentially negative infinity, and those sensors will not be tasked for any more tracks. The *loading_weights* in the weight file give the loading weights in five-percent intervals. For example, the current weight file gives a score of 100 to those sensors that have 0 - 5 percent of their capacity used.

Probability of Acquisition Weight. The final weight to be scored is the probability of acquisition weight, or PA weight. This weight includes a percentage number specific to every object and sensor pair. These numbers are generated from feedback from the SSN, and are kept in the Prototype Tasker's database. This percentage is generated by the number of successful tracks received by the sensor on the object divided by the number of tracks the sensor has been tasked to do on that object. The numbers used in calculating PA span the period of time that the Tasker has been used operationally. For an example of a calculation, let us consider object #1 and sensor #1. In the past, sensor #1 has been tasked to do 1000 tracks on object #1. It has returned 900 successful tracks on object #1. Thus, the PA number for this object and sensor pair is 0.90. The total PA weighting score is:

$$\textit{probability of acquisition weight} = \textit{PA percentage} \times \textit{PA\_weight}$$

The *PA_weight* is found in the weight file.

2.4 Summary

Though the Prototype Tasker has been used successfully for some time, there is still room for improvement (8; 22). One problem is that the SSN is not responding to the taskings as well as it could be. Not all of the tracks that are tasked are performed. Though it is impractical to expect 100 percent efficiency out of the SSN, it is believed that the current response can be improved. The experts believe that sensors may be tasked for objects that they may have difficulty in detecting. This may explain why the SSN response is lower than expected.

The other problem lies in the lack of resources. There are a limited number of sensors that can track deep-space objects, while there are more that are able to track near-earth objects (13:33; 8). The boundary between near-earth and deep-space objects is an orbital period of 225 minutes (13:33; 8). This boundary corresponds to an altitude of about 5000 km for a circular orbit (13:33). The result of the lack of resources is that many of the lower-priority deep-space objects do not get observations as often as they should.

III. Theoretical Background

This chapter reviews the methods used in performing the research. The first section gives a background of the general Simulated Annealing algorithm. The second discusses Adaptive Simulated Annealing, the Simulated Annealing algorithm used in the research. The third section is a review of Multicriterion Optimization and the specific techniques used in the research.

3.1 Simulated Annealing

The purpose of this research was to find the weighting scheme that produces the best tasking from the Prototype Tasker. Chapter 1 gave a short description of how the problem was modeled. After the model of the problem was formulated, an iterative search was chosen as the means for finding solutions. After weighing different types of searches, it was decided to solve the problem with Simulated Annealing (SA).

SA is an iterative search algorithm and has been used in numerical optimization for many different functions and problems. SA has been applied to numerous different applications with excellent results. Our problem can be viewed as finding appropriate parameters for input into an algorithm. The literature does not contain examples of SA's application to this problem.

3.1.1 Introduction of Simulated Annealing (28). Traditional iterative search methods are able to solve many types of problems. However, it can be difficult applying them to certain problems. The main drawback to these methods is that the objective function must be continuous and should contain only one optimum. If the objective function contains more than one, then there is a possibility that the search method will find only a local optimum. The final solutions found by these search methods may vary depending on the initial conditions of the problem.

Iterative search methods “move” from one point to another in the variable space. The algorithm is initialized with a starting point, $\mathbf{x}_i$. A stopping criterion is defined. The stopping criterion is usually represented as a required precision of the objective function. Next, a point, $\mathbf{x}_j$, in the neighborhood of $\mathbf{x}_i$ is found. If the value of the objective function at $\mathbf{x}_j$ is better than that of $\mathbf{x}_i$, then $\mathbf{x}_j$ is “accepted.” This means that that $\mathbf{x}_j$ replaces $\mathbf{x}_i$ and the search continues from this point. If $\mathbf{x}_j$ is less optimal than $\mathbf{x}_i$, then the search continues with finding another point in the neighborhood of $\mathbf{x}_i$. This process continues until the difference in the objective function at consecutive iterations is less than the required precision.

There are several ways to get around the problem of finding only local optimum points when employing an iterative search (28:4). First, the problem can be solved many times with different initial conditions. This can be time consuming, and still may not find the global optimum. Another choice may be to use the data from past solutions to pick better initial conditions. A third method is to use a complicated function for generating the next point the algorithm will consider. The final option is to accept points that have a “worse” objective function value. In a minimization problem, this would mean that points with higher values for the objective function may be accepted. This probabilistic hill climbing (23:9) is the difference between SA and other search methods. It enables the search to escape local minimums.

SA was developed independently by Cerny (2) and by Kirkpatrick, Gelatt, and Vecchi (19). The SA algorithm has its roots in statistical mechanics. The developers of the algorithm found a similarity between the annealing process of matter and the process of finding solutions to combinatorial optimization problems (19:671). When annealing matter, the material is first heated. It is then slowly cooled until it reaches its ground energy state. The temperature of the material cannot be dropped too quickly. This may lead to the material settling in a higher energy state.

The cooling is done in stages. At each stage, the material is allowed to settle to thermal equilibrium. After equilibrium is reached, the temperature is dropped. This process is continued until the ground energy state is reached.

At each temperature, each energy state of the system has a probability of being the equilibrium energy (28:7). This probability is given by the Boltzmann distribution, also known as the Gibbs distribution:

$$\Pr\{\overline{E} = E\} = \frac{1}{Z(T)} \exp\left(\frac{-E}{k_b T}\right)$$

Here, E is the energy of the state, k_b is the Boltzmann constant, T is the temperature, and $Z(T)$ is a normalization factor at temperature T. As the temperature decreases, the only state with a non-zero probability is that with the lowest energy.

A Monte Carlo method described by Metropolis et al. simulates the actual annealing of matter (28:8). One randomly chosen particle is slightly displaced. This gives a new state of the system. If the new state results in a lower energy, then the method continues with the new state. If not, the new state is accepted with probability:

$$\exp\left(\frac{-\Delta E}{k_b T}\right)$$

After a large number of changes in the system, the probability distribution of the states approaches the Boltzmann distribution, which shows equilibrium.

By dropping k_b and substituting a merit or cost function for E, the Metropolis algorithm becomes an optimization method. A set of Metropolis algorithms is sequenced, each with a decreasing temperature. At each level, the algorithm is allowed to reach an equilibrium energy, or cost, state. This procedure results in SA. Temperature is a term still commonly used in SA. In this context it is a control parameter for the probability of accepting a state with a lower merit.

SA has many advantages over other search methods. First, it is sometimes able to find better solutions than other search methods (2:41; 23:7). It is able to handle many kinds of merit functions and constraints (16:29, 35-40). The solutions found are not dependent upon the initial conditions of the problem. In the past, it has been applied to problems in many different fields such as combinatorial optimization and neural networks with excellent results (16:35-40).

3.1.2 The General Simulated Annealing Algorithm. There are four basic things that are needed to solve a problem by SA (19:675). First, one needs a depiction of the configurations of the system and a method of generating new configurations from the current one. Also needed is a method of measuring the value of a configuration. Finally, an annealing schedule needs to be established to provide a means of decreasing the temperature and guiding the search.

The algorithm for a minimization problem can be described by the steps below (20:2.1 - 2.2; 28:10):

Step 1. Initialize system with starting temperature and configuration of your variables, $\mathbf{x}_i$. Solve for the merit function, C_i, at the initial configuration.

Step 2. Perturb the values of one or more of the variables of $\mathbf{x}_i$. This will give you a point, $\mathbf{x}_j$, in the neighborhood of your previous point. Evaluate the merit function, C_j, at this point.

Step 3. If $(C_j - C_i) \leq 0$, then accept $\mathbf{x}_j$. Go to Step 5.

Step 4. If $(C_j - C_i) > 0$, then accept $\mathbf{x}_j$ if $\exp\left(\frac{-(C_j - C_i)}{T}\right)$ is greater than a random number between 0 and 1.

Step 5. If $\mathbf{x}_j$ was accepted, then $\mathbf{x}_i$ is replaced by $\mathbf{x}_j$.

Step 6. Return to Step 2 and repeat until sufficiently close to equilibrium. After approaching equilibrium, lower the temperature according to the annealing schedule.

Step 7. Repeat Steps 2 through 6 until a stopping criterion has been reached.

There are two different types of SA algorithms: the homogeneous and inhomogeneous (28:14). In the homogeneous algorithm, the temperature is kept constant until equilibrium is found at that temperature. The temperature is then dropped and another equilibrium is found. Again, the temperature is dropped. This process continues until a global solution is found. In contrast, inhomogeneous algorithms drop the temperature every time there is an acceptance of a new configuration.

Though there is no "correct" value for the initial starting temperature, most experts agree that it should be relatively high. It can be found by taking an educated guess. Also, a trial and error approach can be used. This is done by picking an initial temperature and running the algorithm for a few iterations to see if this value is acceptable (19:675; 2:45). The initial temperature should be large enough to allow the algorithm to accept almost all trial configurations. This is to ensure that the algorithm does not prematurely settle at a local optimum near the initial point (6:211).

There are different annealing schedules that can be used. Some of the more popular schedules are given below (6:211-213; 20:2.5 - 2.6). In these equations, $T(k)$ is the temperature at k, and k is the number of times the temperature has changed. Two constants are represented by C and α. In most cases, C represents the initial value of the cost function.

• Constant	$T(k) = C$
• Arithmetic	$T(k) = T(k-1) - C$
• Geometric	$T(k) = \alpha T(k-1)$
• Logarithmic	$T(k) = C/(\ln (1+k))$

The annealing algorithm continues until it reaches a stopping criterion. Like the annealing schedules, there are a lot of different stopping criteria that can be applied. Depending on the actual problem, some of them may be more fitting than others. Some examples are provided below (6:213; 20:2.6).

- Stop after a fixed number of iterations.
- Stop after the temperature T(k) is sufficiently close to zero.
- When the algorithm approaches a minimum, the changes in the merit function *C* will change very little. If the change is less than some predetermined value, then stop the algorithm.
- If the number of acceptances is low over successive temperatures, stop the algorithm. This can be represented by a ratio of acceptances to trials.
- If the merit function shows no improvement at successive temperatures, stop the algorithm.

3.1.3 Use of Simulated Annealing. Initially, SA was used in solving combinatorial optimization problems. Over the years, it has been used in a multitude of applications. SA statistically guarantees finding a globally optimal solution (16:29). In other words, SA will converge to the optimal solution if given an infinite amount of time and iterations. The algorithm is easy to implement and is very flexible. Overall, it has been found "useful in finding globally good solutions for a large variety of problems" (27:390).

Though SA can be used to solve many difficult problems, it is not the best algorithm for all problems (17:4). It is intended to solve those problems that do not have an efficient solution algorithm. However, if the objective function has a lot of non-linearities or discontinuities, other algorithms may not be suitable or available. In this case, SA may be an algorithm to consider.

3.2 Adaptive Simulated Annealing

Adaptive Simulated Annealing (ASA) is an SA algorithm developed by Dr. Lester Ingber. It was first developed in 1987, under the name of Very Fast Simulated Reannealing (VFSR) (17:3). Since then, Dr. Ingber has been analyzing his algorithm and making improvements over time. The code is available through Dr. Ingber. It is written in the 'C' programming language, and it is compatible with ANSI C and earlier 'C' compilers.

The program is very flexible. It can be used with either integer or continuous variables and with constrained or unconstrained optimization problems. It requires that the variables are bounded. These bounds can be arbitrarily large and are not a hindrance to the application of ASA to most problems. The user can make many kinds of adjustments to the algorithm based on the particular problem. These are performed though pre-compile and program options. However, all of the options do have defaults. Use of the defaults is suggested if you do not know ahead of time what the options should be for a particular problem (17:4).

The algorithm is placed in two modules: the ASA module and the user module. The objective function and constraints are written to the user module. Also, the dimensionality of the problem and the initial value of the variables, called cost parameters, need to be declared in the user module. The algorithm is written for a minimization problem, but it can be adapted to a maximization problem by using the negative of the cost function.

3.2.1 Equations and Distributions (16:47; 17:3; 20:3.4 - 3.5). Dr. Ingber uses an exponential annealing schedule in the ASA algorithm. This schedule is faster than previous implementations of the SA algorithm. It has been shown that even with this fast annealing schedule, the algorithm will eventually converge to the global optimum (16:47).

However, this is only true if a certain generating function is used. This generating function is discussed below.

Generating Probability Density Function. As with all other SA algorithms, the ASA generation function chooses a new point from the neighborhood of the current point. In many other algorithms, only one parameter is perturbed per iteration. In ASA, when the new point is generated, there is a change in all of the cost parameters, not just one. Also, the size of the neighborhood of the current point decreases as ASA gets further in its search. This is done through the use of a parameter annealing schedule. The parameter annealing schedule controls the size of the neighborhood in the parameter space. It will be discussed later in this section.

Let D represent the dimension of the cost parameter space, or the dimension of the problem. The value of cost parameter i at iteration k for $i = (1,\ldots,D)$, is represented by $p^i{}_k$. The value of p^i at the next iteration, $p^i{}_{k+1}$, is found by $p_{k+1}^i = p_k^i + y^i(B_i - A_i)$. A_i and B_i are the lower and upper bounds of the cost parameter respectively. The variable y^i holds a value between -1 and 1 and is given by:

$$y^i = sgn\left(u^i - \tfrac{1}{2}\right)T_i\left[\left(1 + 1/T_i\right)^{|2u^i - 1|} - 1\right]$$

where u^i is a uniform random variable between 0 and 1 and T_i is the temperature of p^i at iteration k. The formulation for the y's is driven by the probability density function of the state space, $g(y, T)$:

$$g(y,\ T) = \prod_{i=1}^{D} g^i\left(y^i,\ T_i\right)$$

and

$$g^i\left(y^i,\ T_i\right) = \frac{1}{2\left(\left|y^i\right| + T_i\right)\ln\left(1 + 1/T_i\right)}$$

where $g^i(y^i,\ T_i)$ is the density function of y^i at temperature T_i for each individual cost parameter. The cumulative distribution function for one cost parameter is:

$$G^i\left(y^i,\ T_i\right)=\frac{1}{2}+\frac{sgn\left(y^i\right)}{2}\frac{\ln\left(1+\left|y^i\right|/T_i\right)}{\ln\left(1+1/T_i\right)}$$

The cumulative distribution shows that the smaller T_i becomes, the greater the probability that y^i will be smaller in magnitude. Thus the permutations will become smaller and smaller.

Acceptance Probability Density Function. The ASA uses the standard acceptance criterion used in most SA algorithms. The new point p_{k+1} is accepted if:

$$\exp\left[-\left(C\left(p_{k+1}\right)-C\left(p_k\right)\right)/T_{\cos t}\right]>U,\quad U\in[0,1)$$

where U is a uniform random variable and T_{cost} is the cost, or objective, temperature at iteration $k+1$.

Temperature Annealing Schedules. There are two temperatures used in the implementation of ASA: the cost temperature and the parameter temperature. Both are exponential in nature. The parameter temperature controls the size of the step when generating a new point. The cost temperature controls the acceptance probability.

The annealing schedule for each parameter's temperature is as follows:

$$T_i\left(k_i\right)=T_{0i}\exp\left(-c_ik_i^{1/D}\right)$$

In this equation, k_i is the number of iterations performed by ASA and T_{0i} is the initial temperature of parameter i. The constant that controls the rate of annealing is c_i. All c's are given by:

$$c=m\exp(-n/D)$$

There are defaults for the values of exp(-m) and exp(n) in the program options, but they can be changed. The *temperature ratio scale* is the value of exp(-m) and exp(n) is the *temperature annealing scale*. These two parameters control the annealing rate. The effect of these numbers is that at iteration exp(n) the temperature will equal T_{0i} exp(-m). In other words, after the number of iterations set by the *temperature annealing scale*, the temperature will decrease by the factor given by the *temperature ratio scale*. For example, the default *temperature ratio scale* is 10^{-5} and the default *temperature annealing scale* is 100. If the initial temperature is 2, at the 100th iteration, the temperature will be $2\text{x}10^{-5}$.

The cost temperature is the same as the parameter temperatures except for two things. First, the index k in the cost temperature is the number of acceptances, not the number of iterations. Therefore the cost temperature drops only when a new point is accepted. Second, there is another constant that is included in c:

$$c = \text{cost parameter scale} * m \exp(-n / D)$$

The cost parameter scale is another constant that can be chosen through the program options. Its default is 1.

If the user wishes to use different annealing schedules, this can be done. A program option can be set to allow the user to write his own annealing schedule in the user module of the code.

3.2.2 Initial Conditions. The initial conditions of the ASA program include the initial cost and parameter temperatures and the initial values of the cost parameters. For the values of the cost parameters, it is suggested to input a known feasible solution in the user module (17:22). This is not a necessity, since if not given an initial parameters array, ASA will generate random parameters until an initial feasible solution is found. However, Dr. Ingber feels that an initial point chosen by a person cannot do any worse than a randomly chosen point from the computer. The solutions found by ASA are not dependent on initial values of the cost parameters. Inputting an initial feasible solution will, however, save some computation time.

The initial temperatures can be changed by the user, if desired. The default initial parameter temperature is 1. The default initial cost temperature is a little more involved. Before the algorithm begins, ASA finds a number of random points in the parameter space and evaluates the cost function of each one. The average of these cost functions is then the initial cost temperature. In most cases, this number is large enough to ensure that most of the new points are accepted at the beginning of the algorithm. However, the user can specify his own initial cost temperature, if desired.

3.2.3 Stopping Criteria. There are several program options that give the user the opportunity to define the stopping criteria for the problem. All of the numbers can be changed and most can be set to have no effect. They are as follows:

- *Limit Acceptances (default =10000).* ASA will exit and return a final result if the number of acceptances goes over the number set in *limit acceptances*.

- *Limit Generated (default = 99999).* ASA will exit after the number of generated states exceeds *limit generated*.

- *Accepted to Generated Ratio (default = 10^{-6}).* When near the global minimum, the number of acceptances will most likely be very low. The accepted to generated ratio gives a measure to test this logic. When the ratio drops below the defined *accepted to generated ratio,* the algorithm will end.

- *Cost Precision (default = 10^{-18}) and Maximum Cost Repeats (default = 5).* ASA will exit if the cost function repeats at consecutive points. With the default value, if five consecutive points give the same cost function, then the algorithm will stop. *Cost Precision* sets the precision of the cost functions for this stopping criterion.

The algorithm also stops when the temperature drops to "machine" zero. However, the user can bypass this and still continue with a temperature of value zero. However, the algorithm becomes a traditional iterative search method at this point.

3.2.4 Features of ASA (16; 20:3.7 - 3.8). ASA is a very flexible algorithm that can be adapted to many applications. In addition to the flexibility, it does offer some features that are not shared by all SA algorithms. They are self-optimization, reannealing and simulated quenching.

Of these features, reannealing was the only one used in the research. Reannealing is the process of resetting the cost and parameter temperatures (16:48). The default settings cause it to be performed every 100th iteration. In reannealing, the cost temperature schedule is rescaled according to new minima found by the algorithm. If a new minimum is found, it replaces the value of the initial cost temperature. The cost annealing schedule is then reevaluated as if the minimum cost were the initial cost temperature. The algorithm continues from the point it was at before entering reannealing. However, the temperature at the next iteration is changed as if the value of the new minimum was the initial temperature. Reannealing for the parameter temperatures employs the derivative of the cost function with respect to each one of the cost parameters. Since parameter temperature reannealing should not be used with discontinuous cost functions, it was not used in the research.

The ASA algorithm always keeps track of the best point found. Therefore, if the algorithm "settles" on a different point than the optimum, ASA will still return the optimum as the best point.

The ASA algorithm has been used on many different problems and applications (16:35 - 40). Due to its flexibility and availability, it was chosen as the algorithm used to do this research.

3.3 Multicriterion Optimization

The objective function of the research is to maximize the merit of a tasking made by the Prototype Tasker. There were two criteria identified as to how the merit of a tasking may be found. This leads to the possibility of having more than one objective function for the problem at hand. Multicriterion Optimization (MCO) techniques handle problems with more than one objective function. Most of these techniques involve transforming the set of objectives into one objective function.

The standard MCO problem is as follows:

$$\begin{array}{l}\text{Min (Max) } Z_1(\mathbf{x}),\ Z_2(\mathbf{x}),\ \ldots,\ Z_p(\mathbf{x}) \\ \text{subject to: } \mathbf{x} \in B\end{array}$$

This is the same as a standard optimization problem in that we are trying to find a point in a feasible region that optimizes the objective function. The difference here is that there is more than one objective function.

One of the key concepts in MCO is the set of non-dominated solutions. The set of non-dominated solutions, S, is described as (15:20):

$$S = \left\{ \begin{array}{l} \mathbf{x} : \mathbf{x} \in \mathbf{X}, \text{ there exists no other } \mathbf{x}' \in \mathbf{X} \text{ such that} \\ Z_q(\mathbf{x}') < Z_q(\mathbf{x}) \text{ for some } q \in \{1,2,\ldots,p\} \\ \text{and } Z_k(\mathbf{x}') \le Z_k(\mathbf{x}) \text{ for all } k \ne q \end{array} \right\}$$

In other words, a point is in the non-dominated solution set if there are no other points that are at least as 'good' as it in all objectives and it is better in at least one. When moving from one solution to another in the non-dominated solution set, if one objective decreases, then a different one will increase.

Making decisions and optimizing over several objectives is not an exact science. There are techniques that can reveal all possible answers for the problem. In the end, some decision maker has to choose which objective is most important and what kinds of trade-offs need to be made.

Although there are techniques that can find the whole non-dominated solution set, these techniques are limited to linear problems, or simpler integer and non-linear problems. In cases when the behavior of the objectives is quite unknown and possibly erratic, the best an analyst may do is to interact with the experts and decision makers (4). With their help, the analyst may be able to find some solutions that are in agreement with their preferences.

In the research, the objective function was modeled in three ways. Two of the models were derived from two different criteria representing the "goodness" of the tasking. Therefore, MCO was needed to provide an objective function from the two criteria. The two methods used, the weighted sum method and the constrained feasible region method, are discussed below.

3.3.1 Weighted Sum Method. One way to deal with more than one objective is to place a weight on each objective and add them. The objective function then becomes:

$$\text{Min (Max) } Z(\mathbf{x}) = \lambda_1 Z_1(\mathbf{x}) + \lambda_2 Z_2(\mathbf{x}) + \ldots + \lambda_p Z_p(\mathbf{x})$$

$$\text{subject to: } \sum_{i=1}^{p} \lambda_i = 1, \qquad \lambda_i \in [0,1] \quad i = 1,\ldots,p$$

where the λ_i is the weight placed on objective i. For any chosen set of λs, the problem can then be solved by single objective means. It will produce a non-dominated optimal solution, but only for that set of weights. In order to do a complete analysis of the problem, it would be better to find the whole non-dominated solution set. A simple way to do this is to vary the weights. However, there are an infinite number of weights that can be applied to the objectives, so this analysis would be incomplete. For different classes of problems there are algorithms that produce the whole non-dominated solution set and determine for what range of λs each solution is optimal. For example, the Multicriterion Simplex Algorithm produces such an answer for linear MCO problems (3).

3.3.2 Constrained Reduced Feasible Region Method. Another way to deal with multiple objectives is to make all objectives but one a constraint. This is done by setting the objective function less than (or greater than for maximizing) some satisficing level. This will solve for the optimum at those particular satisficing levels but will not give the whole non-dominated solution set. As in the weighted sum method, each satisficing level would need to be varied across all possibilities to construct the non-dominated solution set.

3.3.3 The Use of MCO in the Research. The objective functions used in this research were very complex. Both of the methods mentioned above were used in formulating the objective functions. Determining the whole non-dominated solution set is beyond the scope of the research. For this research, the sponsor gave some ideas as to what was important in the merit functions and what aspects of the taskings could be improved. For example, the λs for the weighted sum approach were chosen to best reflect what the sponsor thought was important and to what degree. This approach produces results that improve the daily tasking and take into account the expressed preferences of the sponsor.

IV. Methodology

4.1 Model of the Prototype Tasker

When attempting to solve a problem by mathematical means, one must first express the problem as a mathematical model. A model is a representation of the problem in mathematical terms. The model is used to help determine the correct approach to the problem. Chapter 1 gave an overview of how the Prototype Tasker was modeled. Figure 4.1 gives a pictorial view of this model.

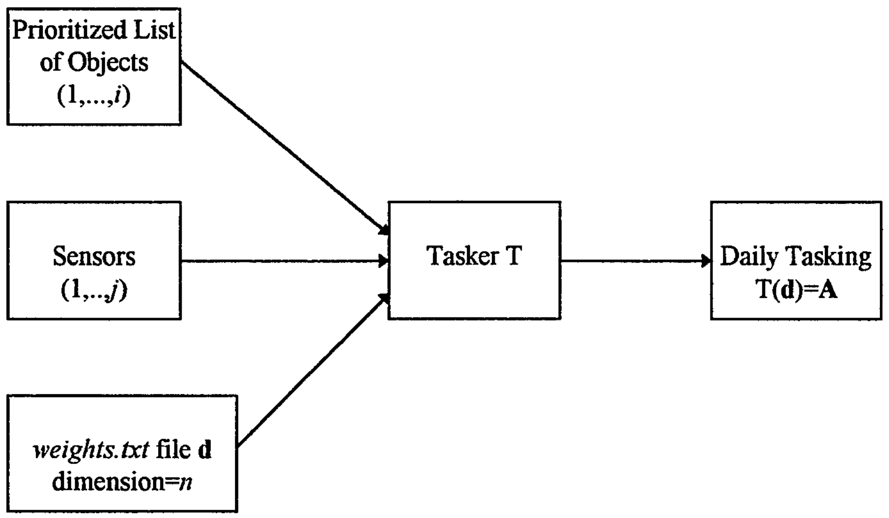

Prioritized List of Objects (1,...,*i*)
Sensors (1,..,*j*)
weights.txt file **d** dimension=*n*
Tasker T
Daily Tasking T(**d**)=**A**

Figure 4.1 Model of the Prototype Tasker

There are three inputs to the Prototype Tasker: the list of prioritized objects, the list of available sensors, and the weight file. The object and sensor lists are kept constant. The weight file, or **d**, is an n-dimensional vector of positive integers. For the problem at hand, n is equal to six. The vector **d** is the set of variables for the problem. The probability of acquisition (PA) and orbit distribution values in the weight file are actually coded as real numbers. Nevertheless, in this research they were restricted to integer values. This allows all of the variables to be of the same type and also reduces the variable space. The Tasker takes these inputs, **d**, and performs function T. The result of this function is the daily tasking, T(**d**) = **A**. The tasking **A** is an *i* x *j* matrix with the rows representing the objects and the columns representing the sensors (Figure 4.2). Each entry in **A** is an integer representing the number of tracks performed for an object and sensor pair.

$$\begin{array}{cc} & \text{sites}(1,\ldots,j) \\ \text{objects } (1,\ldots,i) & \begin{bmatrix} 1 & 0 & \ldots & 2 & 0 \\ 1 & 1 & \ldots & 0 & 0 \\ \vdots & & & & \vdots \\ 1 & 0 & \cdots & 0 & 2 \\ 0 & 0 & \ldots & 0 & 1 \end{bmatrix} \end{array}$$

Figure 4.2 The Tasking Matrix

A merit function M is applied to matrix **A**. The merit function will perform a mapping of **A** onto the set of real numbers, where $Z_+^{i \times j}$ is an $i{\times}j$ matrix of positive integers:

$$\mathrm{M}(\mathbf{A}):\ Z_+^{i \times j} \rightarrow \Re$$

Since M(**A**) is equal to M(T(**d**)), the merit function can be seen as transforming the values in the weight file, an *n*-dimensional vector of positive integers, into a real number:

$$\mathrm{M}(\mathbf{A}) = \mathrm{M}(\mathrm{T}(\mathbf{d})):\ Z_+^n \rightarrow \Re$$

This model brings to light several difficulties with the problem. First, the domain, **d**, is integer. Second, there is no analytical form of the functions T and M. Therefore, the only piece of information that is known about the merit function, M, is its value for any given set of integers **d**. Lastly, M is believed to be discrete and discontinuous. As mentioned in Chapter 1, a small change in the weighting scheme may not change the tasking, A. If the tasking is the same, then the merit function, M, will not change either. However, a large change of the values in the weighting scheme probably will change the tasking, and therefore cause a "jump" in the merit function M.

When solving the problem with ASA, the cost parameters are the values in the weight file and the objective function is the merit function M. The cost parameters need to be maintained and updated in the weight file so that the Prototype Tasker can access them. Therefore, within the definition of the cost function in ASA, the cost parameters need to be written to the weight file. Each evaluation of the objective function will be performed on the same set of data in the Prototype Tasker. The only varying values are the numbers in the weighting scheme.

4.2 The Merit Function

During the research, three merit functions were used. Each one was determined in consultation with the sponsor. A "good" tasking according to the sponsor is one that gets as many tracks on the tasking with as high a probability of acquisition for each track as possible (8). These are the two criteria used to build the merit functions. The sponsor had a slightly higher preference for getting tracks on the tasking. He also placed the same amount of importance on getting more tracks on the tasking and getting an average probability of acquisition of eighty percent (10).

Number of Tracks. For each tasking, it makes sense to try to task as many tracks as possible. Therefore, the number of tracks is one of the criteria used in the merit functions. Not all tracks that are required for the day are tasked to a sensor. This is due to the capacity limitations on each sensor and the fact that the Prototype Tasker will not overload a sensor. This places a constraint on the number of tracks placed on the tasking. This criterion is not calculated as a strict number, but as a percentage. The ratio of the number of tracks on the tasking (*# tracks on tasking*) to the total that were required (*total # of tracks required*) needs to be measured:

$$\%\ tracks\ tasked = \frac{\#\ tracks\ on\ tasking}{total\ \#\ of\ tracks\ required}$$

The *% tracks tasked* gives a percentage of tasked tracks out of those required. This criterion was put into percentage form in order to be able to compare this number more easily to average probability of acquisition. Also, the percentage better represents this goal and provides more information than the *# tracks on tasking*.

Probability of Acquisition. The second criterion is to keep the probability of acquisition (PA) of each track on the tasking as high as possible. PA gives the probability of a sensor getting a good track on the object once the track has been tasked to that sensor. PA is defined in the database of the Prototype Tasker for every object and sensor pair. In order to consider the whole tasking, the average PA for the tracks on the tasking was found:

$$avg\ PA = \frac{\sum_{\text{tracks on tasking}} PA}{\#\ tracks\ on\ tasking}$$

As mentioned above, the PA for each object and sensor pair is kept in the Prototype Tasker's database. A running sum of the total PA is kept current in the Prototype Tasker when each track is assigned. However, these PA values are used in one of the weighting scores in the sensor selection portion of the Prototype Tasker. Therefore, we expect the value of the *PA_weight* in the weight file to be high when solving the problem with ASA.

Merit Function 1. The first merit function used the weighted sum approach:

$$\text{Max } Z = \lambda_1\ (\%\ tracks\ tasked) + \lambda_2\ (avg\ PA)$$
$$\text{where } \lambda_1 = 0.55 \quad \lambda_2 = 0.45$$

These λs were chosen for two reasons. First, the sponsor placed greater importance on the percentage tasked than the PA. Secondly, he placed the same amount of importance on getting tracks on the tasking and getting an overall PA of eighty percent. If *% tracks tasked* and *avg PA* can be compared equally, the λs can be solved:

$$\begin{gathered} \lambda_2 = 0.8\lambda_1 \\ \lambda_1 + \lambda_2 = 1 \qquad \lambda_1, \lambda_2 \in [0,1] \\ \Rightarrow \lambda_1 \cong 0.55 \end{gathered}$$

Merit Function 2. The second merit function uses the constrained feasible region method. The objective is to maximize *% tracks tasked* while *avg PA* acts as a constraint:

$$\begin{aligned} &\text{Max } Z = \ \% \ \textit{tracks tasked} \\ &\text{subject to:} \\ &\quad \textit{avg PA} \ \geq \ r \end{aligned}$$

To find the value of *r*, the Prototype Tasker was run with the currently used, or default, weight file. The *avg PA* from that tasking was used as the value of *r*. This was to ensure that the solutions found by ASA would improve *% tracks tasked* without penalizing *avg PA*.

Merit Function 3. Another criterion to consider is the expected number of tracks that will be performed by the SSN for that daily tasking. The expected value of a random variable, *x*, is given by:

$$\mathrm{E}[x] = \sum_{i=1}^{k} x_i p(x_i)$$

where x_i is a particular value of the random variable and $p(x_i)$ is the probability that $x = x_i$. For each individual track assigned to the tasking, the PA value is the probability of the sensor completing that track successfully. Therefore, for each object and sensor pair, the expected value of the tracks performed equals $(\#\ \textit{tracks assigned}) * \text{PA}$. In order to find the expected number for the whole tasking, all of the expected values for each object and sensor pair are totaled:

$$\sum_{l=1}^{j}\sum_{k=1}^{i}(\#\ \textit{tracks for object } k \textit{ on sensor } l) * (\mathrm{PA}_{object\,k,\ sensor\,l})$$

$$= \sum_{\text{tracks on tasking}} \mathrm{PA}$$

$$i = \text{total \# of objects}, \quad j = \text{total \# of sensors}$$

Therefore, the expected value of tracks that will be successfully performed is the numerator of the *avg PA* statistic.

Since the Prototype Tasker reports *avg PA* rather than total PA, some post analysis had to be performed. In ASA, the third merit function used was:

$$(\%\ \textit{tracks tasked}) * (\textit{avg PA})$$

Since the Prototype Tasker was already configured to write *% tracks tasked* and *avg PA* to log files, these two values were used in the computation of this merit function. This will give the expected number of tracks multiplied by a constant:

$$(\%\ \textit{tracks tasked}) * (\textit{avg PA})$$

$$= \left(\frac{\#\ \textit{tracks tasked}}{\textit{total \# tracks required}}\right) * \left(\frac{\sum_{\text{tracks on tasking}} \mathrm{PA}}{\#\textit{tracks tasked}}\right)$$

$$= \frac{\sum_{\text{tracks on tasking}} \mathrm{PA}}{\textit{total \# tracks required}}$$

Using the same set of data for each run of the Prototype Tasker will keep *total # tracks required* constant. This is another statistic provided by the Prototype Tasker. In post analysis, the *avg PA* can be multiplied by the tracks required to get the true expected value.

4.3 Implementation with ASA

As mentioned in Chapter 3, the ASA code is composed of two modules: the ASA module and the user module. In the user module, the number of variables is declared, initial starting values are set, and the objective function is calculated. The Prototype Tasker model is applied to ASA by setting the values in the weight file to the cost parameters and the merit function to the objective function.

4.3.1 Cost Parameters. The values in the weight file (see Appendix A) that are the variables or cost parameters of the research problem are:

rank_weights
loading_weights
PA_weight
required_no_passes_weight
additional_no_passes_weight
orbit_distribution_weight

Since the weights are relative to one another, one of the parameters can be held constant. It was decided to hold the *orbit_distribution_weight* constant at a value of 5, its default. This is due to its single value. Also, several of the values in the weight file are not included. These values are either not used by the Prototype Tasker or they effect only a small number of objects in the satellite catalog.

The *PA_weight, required_no_passes_weight,* and *additional_no_passes_weight* are ASA cost parameters $x[0]$, $x[3]$, and $x[4]$, respectively. The *rank_weights* and *loading_weights*, also cost parameters, are handled a little differently. As mentioned in Chapter 2, in the weight file, both *rank_weights* and *loading_weights* contain decreasing linear functions (see Appendix A). For the *rank_weights*, the weight will be dependent on the sensor's placement on the sensor ranking list. For example, a ranking of 1 corresponds to a *rank_weight* of 20. A ranking of 2 gives a *rank_weight* of 18. The linear function stops at a ranking of 10, which corresponds to a *rank_weight* of 2. For the *loading_weights*, the currently used capacity of the sensor is used. For example, if 0 - 5 percent of the sensor's capacity is assigned to other objects, then the *loading_weight* for that sensor is 100. If 5 - 10 percent is used, then the *loading_weight* will equal 95. This linear function continues until reaching 100 - 105 percent, where the *loading_weight* is currently equal to 0. For each of these two weights, the value of the cost parameter is the y-intercept of its linear function. The slopes of the linear functions remain constant: -2 for the *rank_weights* and -1 for the *loading_weights*. The cost parameters for *rank_weights* and *loading_weights* are $x[1]$ and $x[2]$, respectively. In order to calculate the actual values written to the weight file, the slope/intercept line formula is used. For both linear functions, the slope is the given constant mentioned above. The values of the cost parameters gives the y-intercept.

The initial values of the cost parameters correspond to the default weight file (see Appendix A). Since these are the values that are currently used with the Prototype Tasker, we would want to start the search with them. The bounds placed on each cost parameter are 0 - 1000. All cost parameters are integer. The cost parameter, its description, and initial value are given in Table 4.1.

Table 4.1 Cost Parameters

Cost Parameter	Description	Initial Value
$x[0]$	*PA_weight*	10
$x[1]$	*rank_weights* y-intercept	22
$x[2]$	*loading_weights* y-intercept	105
$x[3]$	*required_no_passes_weight*	20
$x[4]$	*additional_no_passes_weight*	1

There are several numbers in the weight file that are not included in the cost parameters. The *sensor preference* weights, *currently tasked* weights, and *above/below* *line* weights are not currently being used by the Prototype Tasker. The *selected_weight* and *not_selected_weight* enable the 1 CACS to ensure that an object is tracked by a particular sensor. Due to the nature of this weight and because of the fact that it is used on less than one half of one percent of the objects in the catalog, this weight was ignored. The *max_additional_pass_weight* was always set to the *additional_no_passes_weight*. This was done to stay consistent with the way the *max_additional_pass_weight* is currently used.

4.3.2 The Objective Function. The user module of ASA includes a function called *cost_function* (17:18 - 19). This is the evaluation of the objective function. It is called after a new point is generated by the ASA algorithm. The new values of the cost parameters are passed into this function. Also, the parameters' bounds and first and second derivatives of the objective function with respect to each parameter are passed into *cost_function*. This is to allow access to these values if the user needs them in the development of the objective function. There is also a *valid_state_generated_flag* that can be set equal to false. This is used to declare that a solution violates constraints of the problem.

For the problem at hand, the Prototype Tasker had to be integrated into *cost_function*. The first step in *cost_function* is to write the new weight file. The newly generated cost parameters are used in calculating the new value on each line of the file. After the new weight file is written, a system call is made to execute the Prototype Tasker. At this point, the Tasker performs all of its functions, including generating a new tasking. It also writes the *% tracks tasked* and the *avg PA* values of the new tasking to a log file. After the execution of the Prototype Tasker, ASA then reads the values in the log file. These values are used in the computation of the objective function. Finally, *cost_function* returns the value of the objective function.

4.3.3 Making the Problem Workable. One of the biggest limitations in this research was time. The Prototype Tasker takes about one and one-half hours to complete a tasking on the whole satellite catalog. Since ASA can take several thousand iterations to converge on a solution, it would take several months to make one run on ASA using the whole catalog in the Prototype Tasker.

The Prototype Tasker can be set to produce a new tasking on a subset of the satellite catalog. This subset can be defined by tasking group or satellite numbers. Running the Prototype Tasker on a small set of objects decreases the time needed to make the tasking. For example, a tasking made on 150 satellites takes less than two minutes to complete. When the Prototype Tasker produces a tasking on a subset of the catalog, it removes the objects in the subset from the current tasking stored in the database. After the objects are removed from the current tasking, it then recalculates the capacity available at each sensor (22). The Prototype Tasker then performs a tasking of the satellites in the subset. Those that are not in the subset still have the same tasking as they did before. In other words, the other objects that are not in the subset remain on the tasking with their most current sensor assignment.

The object subsets were chosen to try to give the best representative cross section of the satellite catalog. Two percent of the objects, or around 160 objects, were used. This amounts to one object chosen out of every fifty in the catalog. The objects were chosen according to tasking group. Two percent of the objects in each tasking group were chosen at random. If a tasking group did not contain at least fifty objects, it was combined with other tasking groups with similar tasking characteristics. Out of these groupings, one out of every fifty objects was chosen for the subset.

Running the combined ASA/Prototype Tasker program on a subset of objects made SA workable. However, each run took almost a day to converge to an answer. This fact severely limited the number of times the program could be executed.

V. Results and Analysis

This chapter presents the results of four different executions of the ASA algorithm. The varying elements in the different executions include the merit function, the annealing rate, and the objects used in the optimization. The conditions for each execution are given in the discussion of their results.

5.1 Experimental Conditions

In order to make SA a feasible approach to this problem, only two percent of the satellite catalog was used in the optimization. This amounted to about 160 objects. As mentioned in Chapter 4, this was done to reduce the time of each iteration in ASA. Using the subsets reduced the time of an iteration from over one hour to two minutes. However, when considering thousands of iterations at two minutes each, the computation time becomes quite large. Each execution of ASA completed in seventeen to nineteen hours. Two subsets of the objects were built. The first subset was used in the first execution of ASA and the second subset was used in the other three.

Before the successful executions of ASA were made, a different attempt was made. In this execution, both functions for the *rank_weights* and the *loading_weights* were modeled as:

$$\mathrm{a}x^2 + \mathrm{b}x + \mathrm{c} + \mathrm{d}e^x$$

In this model, 'a,' 'b,' 'c,' and 'd' were the variables, or cost parameters. Therefore, there were four cost parameters for the *rank_weights* and four for the *loading_weights.* The run was made with the default settings of the ASA algorithm (see Appendix B). This execution of ASA never converged and had to be aborted after four days. Also, the intermediate solutions did not produce good taskings. It was then decided to cut the problem by only using 'c' and dropping the other terms. Previous research performed by the 1 CACS and The MITRE Corporation revealed that the shape of the function for each of these weights did not have a large impact on the tasking (22). Therefore, the squared, linear, and exponential terms were dropped. The slope of the weights was kept at the current value, and only 'c,' the *y*-intercept, was used as a cost parameter.

Most of the defaults for the program options in ASA were used. However, some of them were changed in order to reduce the time of the execution. The *accepted_to_generated_ratio* was increased from 10^{-6} to 10^{-3}. This was set so that the algorithm would terminate when one out of every 1000 states generated were not accepted. Second, the *cost_precision* was increased from 10^{-18} to 10^{-7}. During preliminary runs of ASA, it was noticed that the changes in the objective function occurred at the fifth decimal place or higher. Therefore, a precision greater than 10^{-7} would not be needed. The *testing_frequency_modulus* tells how often ASA tests for stopping criteria and performs reannealing. It was decreased from 100 to 50. Since 100 iterations took more than three hours to complete, it was decided to test for the stopping criteria more often than the default allowed. Lastly, the *temperature_ratio_scale* was decreased for the first three executions of ASA. This was done to increase the rate of annealing. The default setting for *temperature_ratio_scale* is 10^{-5}. As mentioned in Chapter 3, the default setting will set the temperature at the 100th iteration to 10^{-5} times the initial temperature. For the first three runs of ASA, this was set to 10^{-6}. Though this may have an impact on the quality of the final solution, it was decided to try to get the algorithm to converge to an answer faster.

Once a solution was found by ASA, this solution was checked against the whole satellite catalog. First, the Prototype Tasker was executed with the current default settings of the weight file. Then it was executed with the file found by ASA. The numbers for *% tracks tasked*, *avg PA* and *expected number of tracks*, were compared. This was done to verify the assumption that using two percent of the catalog would produce good solutions for the whole satellite catalog. In addition to running the ASA solution on the whole catalog, it was also run on the other subset of the catalog. For example, if subset 1 was used in the optimization algorithm, then the resulting weight file was also run on subset 2. This was to verify that a "good" solution for the first subset would also be a "good" solution for another subset of the catalog. Also, a second day of data was provided. The same comparisons were made with this data, when possible.

The computer print outs of the ASA output files and resulting weight files are located in Appendix C. The results of all of the runs are summarized in Table 5.1.

Table 5.1 Results of Each Solution

	Run	default	Run #1	Run #2	Run #3	Run #4
weights	*PA_weight*	10	883	525	10	993
	rank_weights y-intercept	22	186	165	22	729
	loading_weights y-intercept	105	369	283	105	993
	required_no_passes_weight	20	82	59	20	98
	additional_no_passes_weight	1	50	151	1	257
	% tracks tasked	0.798319	0.798319	0.798319	0.798319	0.798319
subset 1	*avg PA*	0.630246	0.663850	0.657480	0.630246	0.657480
day 1	λ_1*% tracks tasked* + λ_2 *avg PA*	0.722686	0.737808	0.734941	0.722686	0.734941
	expected # tracks	299.367	315.329	312.303	299.367	312.303
	% tracks tasked	0.835069	0.831597	0.833333	0.835069	0.833333
subset 2	*avg PA*	0.791267	0.814473	0.813337	0.791267	0.813337
day 1	λ_1*% tracks tasked* + λ_2 *avg PA*	0.815358	0.823891	0.824335	0.815358	0.824335
	expected # tracks	380.599	390.133	390.402	380.599	390.402
	% tracks tasked	0.752665	0.747547	0.748036	0.752665	0.747938
full catalog	*avg PA*	0.698555	0.708217	0.705818	0.698555	0.706658
day 1	λ_1*% tracks tasked* + λ_2 *avg PA*	0.728316	0.729849	0.729038	0.728316	0.729362
	expected # tracks	16128.2	16240.1	16195.7	16128.2	16210.7
	% tracks tasked	0.774914	0.774914	0.776632	0.774914	0.774914
subset 1	*avg PA*	0.569184	0.640845	0.629968	0.569184	0.633411
day 2	λ_1*% tracks tasked* + λ_2 *avg PA*	0.682336	0.738473	0.710633	0.682336	0.711238
	expected # tracks	256.702	289.021	284.746	256.702	285.668
	% tracks tasked	0.773063	0.765683	0.765683	0.773063	0.765683
subset 2	*avg PA*	0.693933	0.736935	0.736065	0.693933	0.736651
day 2	λ_1*% tracks tasked* + λ_2 *avg PA*	0.737455	0.752746	0.752355	0.737455	0.752619
	expected # tracks	290.785	305.828	305.467	290.785	305.710

5.2 Results

5.2.1 Run #1.

The first execution of the ASA algorithm used the weighted sum merit function:

$$\text{Max } Z(\mathbf{x}) = \lambda_1 \text{ (\% tasked)} + \lambda_2 \text{ (avg PA)}$$
$$\text{where } \lambda_1 = 0.55 \quad \lambda_2 = 0.45$$

The optimization was performed over the first subset of the satellite catalog for the first day of data. The annealing rate was made faster by setting *temperature_ratio_scale* to 10^{-6}. All of the other settings in ASA were set as mentioned in Section 5.1.

The solution provided the following cost parameters:

Table 5.2 Cost Parameter Solution for Run #1

cost parameter	definition	ASA solution	default
x[0]	*PA_weight*	883	10
x[1]	*rank_weights y*-intercept	186	22
x[2]	*loading_weights y*-intercept	369	105
x[3]	*required_no_passes_weight*	82	20
x[4]	*additional_no_passes_weight*	50	1

One issue to note is that the *PA_weight* is very high. This happens because each individual PA for each object and sensor pair that appears on the tasking is added in computing the *avg PA* criterion.

The value of the objective function for this solution was 0.737808. This is an increase over the value of the objective function at the default weight file, which is 0.722686. The solution occurred at iteration 88, which was the forty-sixth acceptance of the ASA algorithm. The algorithm ended at iteration 398 because the objective value had repeated five times in a row.

Table 5.1 shows the *% tracks tasked*, *avg PA* and objective value $\left(\lambda_1\left(\%\ \textit{tracks tasked}\right)+\lambda_2\left(\textit{avg PA}\right)\right)$ for both the default weight file and the solution weight file. Though the optimization was performed over object subset 1 for the first day of data, these values were also calculated for subset 2 and the full satellite catalog. In these cases, the Prototype Tasker was run using the weighting scheme generated by the optimization and also the default weight file. Also, the expected number of tracks was calculated for both weight files. This was done to facilitate comparisons of all of the results. The expected number of tracks can also be found in Table 5.1.

All four executions of ASA were performed on data corresponding to Julian date 200. The data for Julian date 201 was also available. Therefore, the above calculations were performed for Julian date 201 data, also. Due to software problems, these results had to be limited to the two subsets of the satellite catalog. The results from the second day are also in Table 5.1.

A couple of things are worth mentioning. First, the solution weight file shows improvement in the merit function for both days and all sets of the catalog. Also, these results show a trade-off: if the *avg PA* is increased, then the *% tracks tasked* has to be decreased. However, the increase in *avg PA* is about an order of magnitude larger than the decrease in *% tracks tasked.* Also, during the optimization, it was noticed that the *% tracks tasked* for the objects in subset 1 never varied. Each iteration did produce a different tasking, since the *avg PA* values changed. However, the same number of tracks was assigned in each tasking.

5.2.2 Run #2. The second run of the ASA algorithm was executed under the same conditions as run #1. The difference is that the second subset of the objects was used in the optimization. Table 5.3 shows the values for the weights found in run #2.

The objective value found was 0.824335, which is an increase from the default value of 0.815358. This value occurred at iteration 20 and was the twelfth configuration accepted. The algorithm exited at iteration 363 due to repeating objective values. All results are in Table 5.1.

Table 5.3 Cost Parameter Solution for Run #2

cost parameter	definition	ASA solution	default
$x[0]$	*PA_weight*	525	10
$x[1]$	*rank_weights* y-intercept	165	22
$x[2]$	*loading_weights* y-intercept	283	105
$x[3]$	*required_no_passes_weight*	59	20
$x[4]$	*additional_no_passes_weight*	151	1

The cost parameters for this run are of about the same relative order of magnitude as those for run #1, except for the *additional_no_passes_weight.* In this case, it is much higher, and is the overall fourth highest variable. Also, in most cases, an increase of *avg PA* required a decrease in *% tracks tasked.* This holds true except for the first subset on the second day. In this case, there is an increase in both *avg PA* and *% tracks tasked.*

5.2.3 Run #3. For this execution, the objective function was the *% tracks tasked* and *avg PA* was used as a constraint:

$$\begin{array}{l} \text{Max } Z = \ \% \ \textit{tracks tasked} \\ \text{subject to:} \\ \quad \textit{avg PA} \ \geq \ r \end{array}$$

This form of the objective function was run against subset 2 for the first day of data. As mentioned in the results of run #1, there was no change noticed in the *% tracks tasked* in subset 1. Therefore, maximizing *% tracks tasked* on the first subset would be unproductive since the objective value showed no change. The value of *r* was picked from the value of *avg PA* found when subset 2 was solved with the default weight file. The *avg PA* was found to be 0.791267, but *r* was dropped to 0.791266 to allow for rounding errors.

The solution returned from ASA was the default weight file. In other words, ASA was not able to find a better set of cost parameters that increased the *% tracks tasked* without decreasing the *avg PA*. It seems as though ASA left the initial starting point for a point in the parameter space with a value less than that of the initialization point. It then returned to the initial point after 288 iterations. The algorithm continued searching until 557 iterations were completed. It exited due to repeats in the objective function.

5.2.4 Run #4. In this execution, the second subset of the objects for the first day of data was used. The objective function was a measure of the number of tracks expected to be performed from the tasking produced: $(\%\ \mathit{tracks\ tasked}) * (\mathit{avg\ PA})$. In the actual implementation, the objective function was multiplied by both 0.45 and 0.55. This explains the lower than expected numbers for the objective function in the ASA output. In post analysis, the *avg PA* was multiplied by the *# tracks on tasking* to get the expected number of tracks that would be performed from that tasking.

The annealing rate for this run was left at the default, 10^{-5}. This was done to see if the algorithm would converge in a reasonable time with the default. This will raise the confidence level that ASA has found a 'good' global solution. The best solution was found on the 45th iteration, which was the 33rd acceptance. The algorithm ended at iteration 381 with repeat cost as the stopping criterion. It converged in about the same number of iterations as the first and second runs. The objective found was 0.1677057, which corresponds to 390.402 expected tracks. This is an increase over that found for the default weight file, which gives a value of 380.599. The cost parameters found at the best point are given in Table 5.4.

Table 5.4 Cost Parameter Solution for Run #4

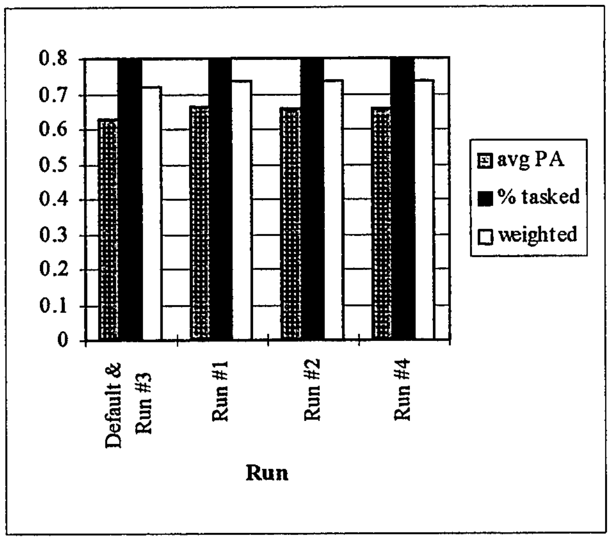
cost parameter	definition	ASA solution	default
$x[0]$	*PA_weight*	933	10
$x[1]$	*rank_weights y*-intercept	729	22
$x[2]$	*loading_weights y*-intercept	933	105
$x[3]$	*required_no_passes_weight*	98	20
$x[4]$	*additional_no_passes_weight*	257	1

There are several interesting outcomes worth mentioning. First, all of the resulting cost parameters are much higher in this run than any of the others. Also, the cost parameters associated with *PA_weight* and the *loading_weights* are set at the same value. This shows that with this particular merit function, these weights are of equal importance. Second, the values of *% tracks tasked* and *avg PA* for this run and run #2 are exactly the same for both of the subsets. This means that the same tasking was found with two different weight files. This shows the existence of alternate solutions that produce the same merit value. The reason why the two runs came up with different weight files could be due to either the objective function or the annealing rate.

5.3 Analysis

In this section, the results of the four runs will be compared, along with the current default settings used on the Prototype Tasker. In all of the runs except the third, the *avg PA* was increased, but at the expense of *% tracks tasked.* The resulting merit functions were very similar in their results. Though there were changes in *% tracks tasked* and *avg PA*, they were not extremely large. This can be due to three different things. First, the default weight file may have been a good solution to the original problem. Second, the weight file may not be as influential as once thought. Third, only about twenty percent of the objects are retasked daily. Therefore, the differences in the merit functions are caused by a small percentage of the objects in the satellite catalog.

The results indicate that *avg PA* can be increased by several percentage points daily. This can be accomplished with a small decrease of *% tracks tasked.* The *expected # tracks* verifies that this will produce a positive effect on the taskings produced. Figures 5.1 through 5.5 show charts of the changes in the merit functions across the different object subsets and weight files. In these charts, the bars signifying "weighted" represent the values of the weighted sum of *% tracks tasked* and *avg PA*.



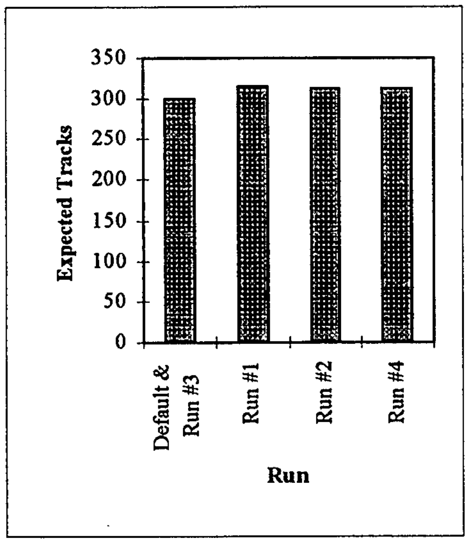


Figure 5.1 Merit Functions for Subset 1 on Day 1

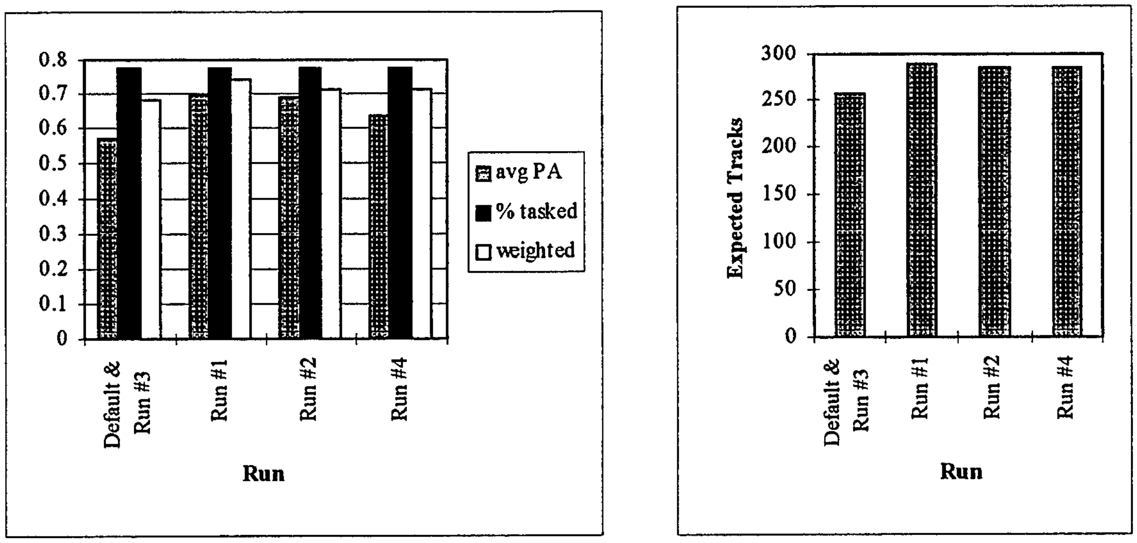


Figure 5.2 Merit Functions for Subset 1 on Day 2

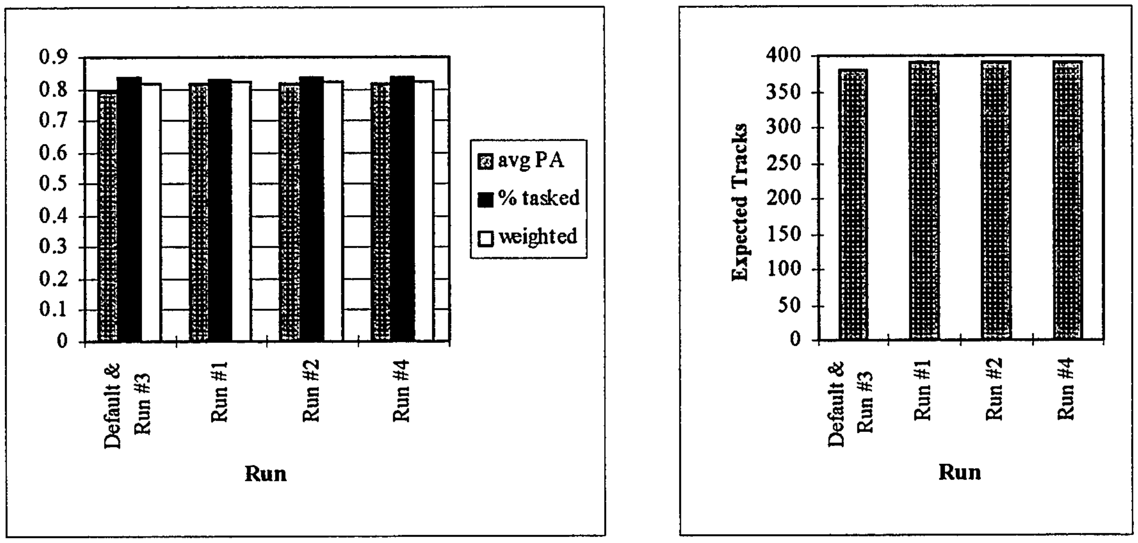
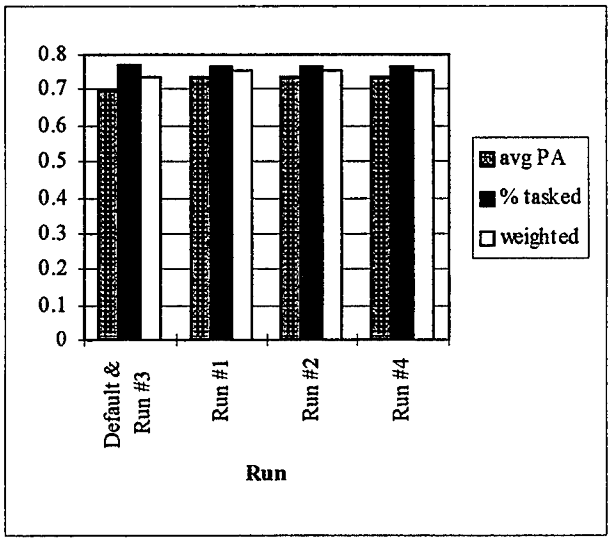


Figure 5.3 Merit Functions for Subset 2 on Day 1



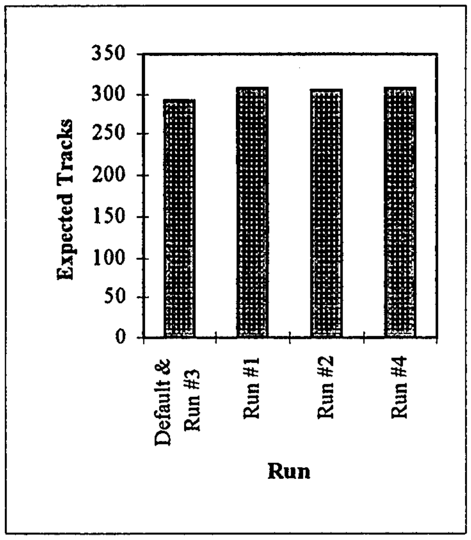
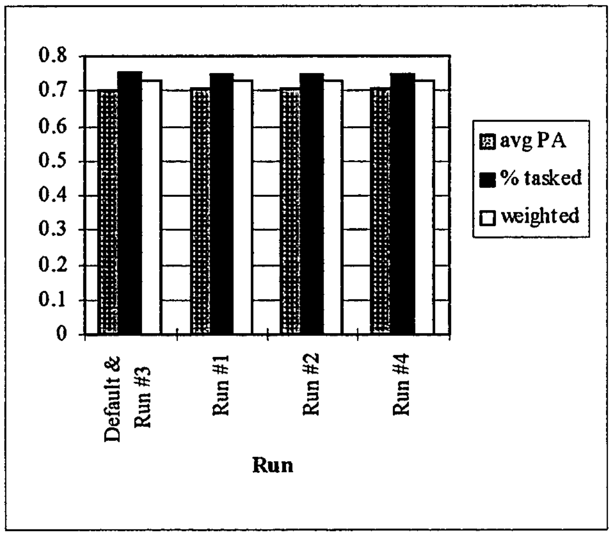


Figure 5.4 Merit Functions for Subset 2 on Day 2



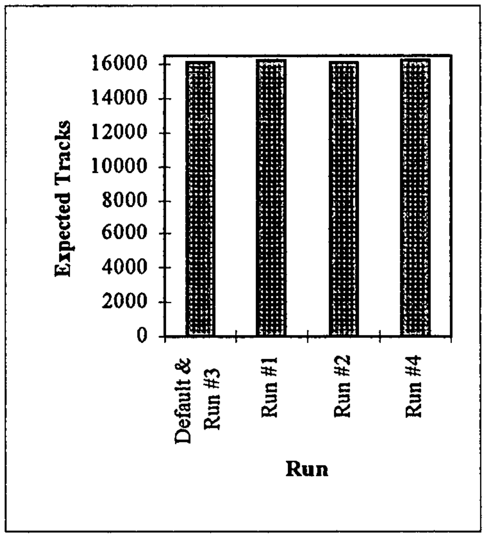


Figure 5.5 Merit Functions for the Full Catalog on Day 1

The disturbing thing about the results is that the values in the weight files were quite different. Figure 5.6 shows the fluctuation of the cost parameters for the default and the solutions found in each run. However, all of the cost parameters found by ASA have a trend, except for *additional_no_passes_weight*. The highest weight for all runs is the *PA_weight*. This is followed by the *loading_weights*, then by the *rank_weights*. The second and fourth run gave higher weighting to *additional_no_passes_weight* than to the *required_no_passes_weight*. The opposite is true for the first run.

Some insight can be given as to why certain numbers appeared in the cost parameters. First of all, the *PA_weight* is an integral part of the computation of *avg PA*. Therefore, its value is expected to be very high. The *rank_weights* are derived from a list that the system experts had compiled. From past experience, they tried to rank a list of sensors for each tasking group. The effect of this weight on the chosen merit functions is quite uncertain. The *loading_weights* is a way for the system to prevent overloading the sensors. This constraint was not declared in ASA since the Prototype Tasker already used the *loading_weights* and other constraints to enforce it. Its decreasing linear function allows for sensors with more capacity available to get a higher score. Since this weight constrains the problem, it is not surprising that it is the second largest weight found by ASA. The effect of the *required_no_passes_weight* and *additional_no_passes_weight* is obscure. In order to really measure their usefulness, a study would need to be made on the operational system or a simulator. These weights have no real input into the merit functions used in this study. These weights were designed to give those sensors with more opportunities to acquire tracks on the objects the assignment of those tracks. Lastly, all of the cost parameters were much higher than those in the default weight file. Remember that the *orbit_distribution_weight* was held constant at five. So it seems that the *orbit_distribution_weight* is far less significant than the other weights. This may be due to the fact that the inverse of the *orbit_distribution_weight* is used in the assign sensor

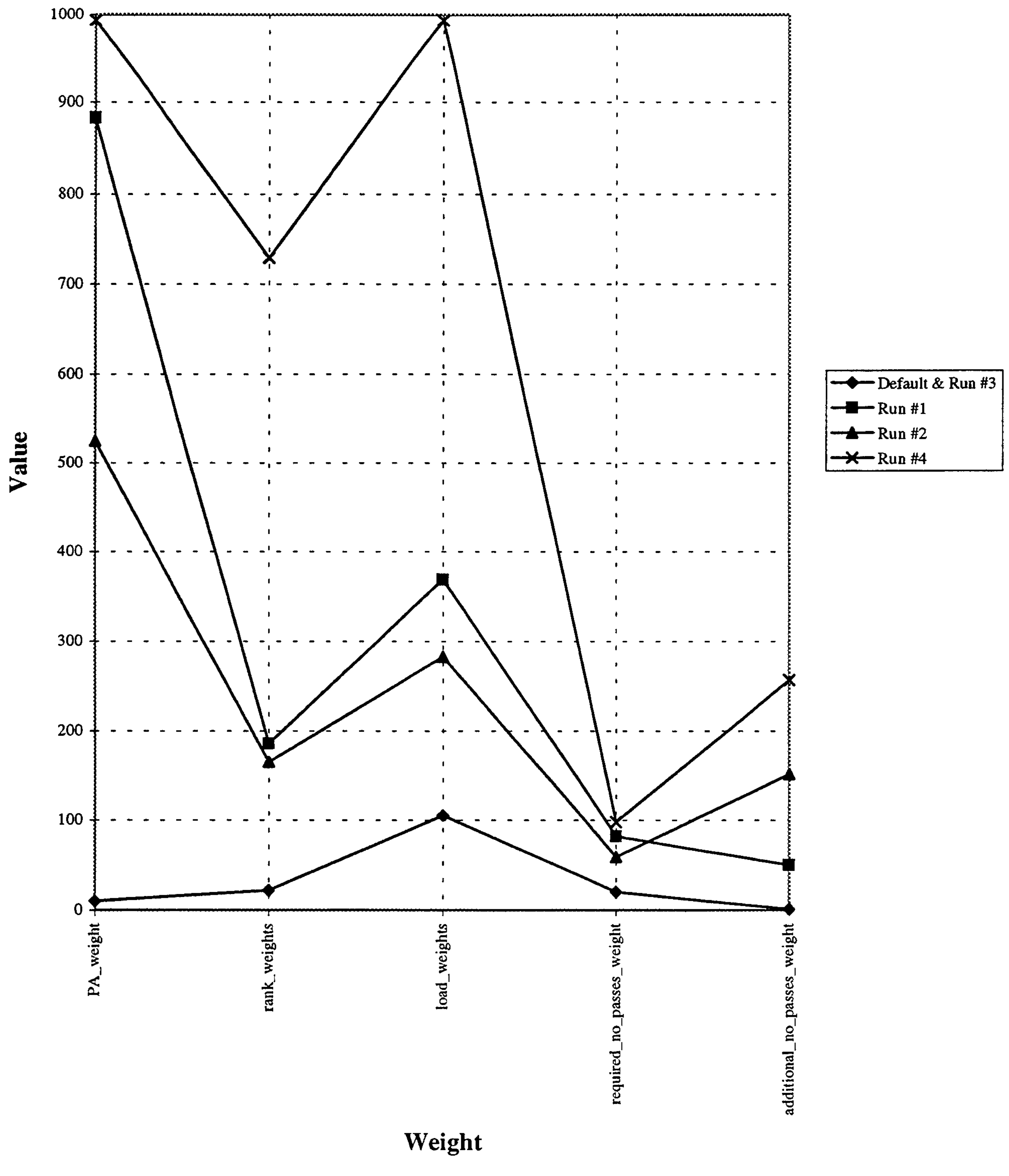


Figure 5.6 Fluctuation of the Weights

portion of the Prototype Tasker. In other words, the lower it is, the more significant it is. However, the effect of the *orbit_distribution_weight* was not included in the merit functions used and its effect can only be seen in the quality of the satellite catalog.

Overall, it is believed that some benefit will come out of changing the weights from the current default settings. However, this research was conducted on two days of data. In addition, the data were for two consecutive days. Since the satellite catalog does not change much on consecutive days, we cannot assume that the data from the two days are independent. It is uncertain what exactly would happen if the research was conducted on other data sets. However, since similar results were seen in the two days, it increases the confidence that the results can be applied to other days.

Another problem is that the long term effects of changing the weight file is unknown. When testing the different weight files on the second day, a tasking from the first day had to be accepted. This was to give the "current tasking" to be used by the Prototype Tasker in the second day. Due to the amount of time needed to reset the values in the database, only one tasking was accepted for the first day. It was the one generated by the default weight file. Therefore, the results for the second day did not reflect the effects of using the same weight file for two days in a row.

The decision for the sponsor is to decide whether to change the weight file or not. Though number of tracks on the tasking may decrease, the probability of the SSN responding with more actual completed tracks increases. In this researcher's opinion, the *PA_weight* should be increased at least to the level of the *loading_weights*. Also, new values of the weight file could be tried. Though other executions may have provided slightly better results, this researcher holds the highest confidence in the results produced by run #4. First, in this researcher's opinion, run #4 had the best merit function. Also, the annealing rate for this run was set at the default level. Therefore, it will be worthwhile to implement results of run #4. If the sponsor is not willing to take this leap of faith, then the

Prototype Tasker can be run without accepting the tasking. This will allow for analysis before the tasking is used operationally. Another option is to use values that have the same relative magnitude of the weights found by ASA. However, the effects of the *orbit_distribution_weight* and the two *no_passes_weights* have not been fully analyzed. Also, since the *selected_weight* was not included in the research, it will need to be rescaled in order to be at least one order of magnitude larger than the other weights.

VI. Summary and Recommendations

6.1 Summary of the Research

The purpose of the research was to find the best set of parameters to use in the Prototype Tasker. These parameters are kept in a user-definable weight file, *weights.txt*. The implementation of the problem involved modeling the Prototype Tasker as a function. One of the inputs of this function is the weight file. The result of the function is a daily tasking for the SSN. A merit function was applied to the tasking to measure the "goodness" of the tasking. The goal was to find the weight file that produces a tasking with the highest possible merit function.

This problem was solved with an iterative search. In particular, a Simulated Annealing algorithm, Adaptive Simulated Annealing (ASA), was used. The values in the weight file were the variables, or cost parameters, in the ASA algorithm. The objective function was the merit function applied to the daily tasking produced by the Prototype Tasker. In order to link ASA and the Prototype Tasker, a system call was made to the Prototype Tasker from within ASA.

The Prototype Tasker takes close to one and one-half hours to compute a tasking on the whole satellite catalog. Since ASA may take several thousand iterations to converge, the time needed to perform a tasking needed to be reduced. Therefore, the Prototype Tasker was configured to perform a tasking on a subset of the satellite catalog, which took about two minutes to complete. The ASA algorithm combined with the Prototype Tasker took almost a day to converge.

Three different merit functions were applied to the problem. Two criteria were used in these merit functions: the percentage of tracks tasked, *% tracks tasked*, and the total average probability of acquisition on the tasking, *avg PA*. Two different subsets of the satellite catalog were used. The values found for the weight file varied quite a bit among the four solutions. However, there was a trend of relative magnitude between these values.

Three major findings were the result of the research. First, the solutions showed a trade-off between the *% tracks tasked* and the *avg PA*. In order to increase *avg PA*, the *% tracks tasked* had to be decreased. Also, the *% tracks tasked* could not be increased without decreasing the *avg PA*. However, it is believed that an increase in *avg PA* is desired. Though the number of tasked tracks will decrease, this is more than compensated for by the increase in *avg PA*. The expected number of tracks actually performed by the SSN will increase. The second finding is that the *PA_weight* in the weight file is a very important value. Making the *PA_weight* the highest value in the weight file will increase the *avg PA* for the tasking. This increases the probability that the SSN will return more observations to the SCC. Third, the values for the objective function did not change drastically. This shows that the currently used weights may have produced a good solution. This may also indicate that the weights are not very influential. Another possible explanation for the small changes in the merit functions is that only about twenty percent of the objects change their tasking for each run of the Prototype Tasker. Though the changes in the merit functions are not drastic, a change in the weights may prove to be significant over time.

The research is not without limitations. Due to the amount of time the software took to compute results, only four runs were executed. Also the sample space was limited to two days of data. As mentioned in Chapter 1, a merit function was used to represent the "goodness" of a tasking. It would be best to use the quality of the satellite catalog or the response of the SSN as a figure of merit. Since a simulator of the SSN does not exist, this approach was not feasible.

Though the actual logic of the Prototype Tasker was not part of the research, some comments should be made. The Prototype Tasker has been used operationally for some time with good results. However, the goal of the research was to find what values in the weight file produce the best solution. It is believed that there may be a more direct approach to solve the problem: change the logic of the Prototype Tasker from a greedy algorithm to a strict optimization algorithm. The sensor allocation portion of the Prototype Tasker can be modeled as an assignment problem that can be solved to optimality. Efficient algorithms for solving assignment problems do exist (1:477-478). Though the complexity of the SSN and the taskings involved may not make this approach feasible, it is still worth considering.

6.2 Recommendations for Future Research

When first approaching the problem, the 1 CACS had proposed several different aspects of the Prototype Tasker that could be researched. For example, research was suggested for finding proper numbers on the tasking tables, mentioned in Chapter 2. Also, the scheduling procedures of some of the sensors in the SSN could be evaluated (10).

Another recommendation is to continue this research. The sample space used here was quite small. The same kind of approach can be used on more data. This will allow for a more complete analysis and also may determine the long term affects of changing the weight file. Also, the problem could be solved with a different approach such as Response Surface Methodology (RSM). However, implementation of RSM requires reasonable knowledge of the domain of the problem, which in this case may not exist. In addition, if a simulator of the SSN does become available, the merit function of the problem can be changed. This would allow for a model of the problem that better reflects the problem at hand.

A third recommendation is to reevaluate the logic of the Prototype Tasker. Any research made in this area may not be implemented quickly. However, it would be useful to know if different logic would produce better results.

APPENDIX A: Current Default *weights.txt* File

```
ORBIT_DISTRIBUTION_WEIGHT
5.0
RANK_WEIGHTS
0
20
18
16
14
12
10
8
6
4
2
0
LOADING_WEIGHTS
005.0                  100
010.0                   95
015.0                   90
020.0                   85
025.0                   80
030.0                   75
035.0                   70
040.0                   65
045.0                   60
050.0                   55
055.0                   50
060.0                   45
065.0                   40
070.0                   35
075.0                   30
080.0                   25
085.0                   20
090.0                   15
095.0                   10
100.0                    5
105.0                    0
110.0              -655360
120.0              -655360
140.0              -655360
160.0              -655360
180.0              -655360
200.0              -655360
SENSOR_PREFERENCE_WEIGHT
0
500
400
300
200
100
0
0
0
0
```

```
PA_WEIGHT
10.0
SELECTED_WEIGHT
1000
NOT_SELECTED_WEIGHT
0
CURRENTLY_TASKED_WEIGHT
250
NOT_CURRENTLY_TASKED_WEIGHT
0
ABOVE_LINE_WEIGHT
250
BELOW_LINE_WEIGHT
0
REQUIRED_NO_PASSES_WEIGHT
20
ADDITIONAL_NO_PASSES_WEIGHT
1
MAX_ADDITIONAL_NO_PASSES_WEIGHT
1
```

APPENDIX B: Default Program Settings for ASA

LIMIT_ACCEPTANCES [10000]
LIMIT_GENERATED [99999]
LIMIT_INVALID_GENERATED_STATES [1000]
ACCEPTED_TO_GENERATED_STATES [1.0E-6]
COST_PRECISION [1.0E-18]
MAXIMUM_COST_REPEAT [5]
NUMBER_COST_SAMPLES [5]
TEMPERATURE_RATIO_SCALE [1.0E-5]
COST_PARAMETER_SCALE [1.0]
TEMPERATURE_ANNEAL_SCALE [100.0]
USER_INITIAL_COST_TEMPERATURE [FALSE]
INITIAL_PARAMETER_TEMPERATURE [1.0]
TESTING_FREQUENCY_MODULUS [100]
ACTIVATE_REANNEAL [TRUE]
REANNEAL_SCALE [10.0]
MAXIMUM_REANNEAL_INDEX[50000]
QUENCH_PARAMETERS [FALSE]
QUENCH_COST [FALSE]

APPENDIX C: ASA Solutions and Resulting *weights.txt* Files

Initial Conditions for Each Execution

ACCEPTED_TO_GENERATED_STATES = 0.001
COST_PRECISION = 1e-07
TESTING_FREQUENCY_MODULUS = 50

index_v	parameter_minimum	parameter_maximum	parameter_value	parameter_type
0	0	1000	10	2
1	0	1000	22	2
2	0	1000	105	2
3	0	1000	20	2
4	0	1000	1	2

Parameter type 2 corresponds to integer variables that are not reannealed.

Run #1

TEMPERATURE_RATIO_SCALE = 1e-06

Results:

```
number_generated = 398,  *number_accepted = 250
best...->cost = -0.737808
best_generated_state ->parameter[0] = 883
best_generated_state ->parameter[1] = 186
best_generated_state ->parameter[2] = 369
best_generated_state ->parameter[3] =  82
best_generated_state ->parameter[4] =  50

COST_REPEATING exit_status = 3
*number_accepted at best_generated_state ->cost = 46
*number_generated at best_generated_state ->cost = 88
```

Run #1 Resulting *weights.txt* File

```
ORBIT_DISTRIBUTION_WEIGHT
5.0
RANK_WEIGHTS
0
184
182
180
178
176
174
172
170
168
166
164
LOADING_WEIGHTS
005.0                    364
010.0                    359
015.0                    354
020.0                    349
025.0                    344
030.0                    339
035.0                    334
040.0                    329
045.0                    324
050.0                    319
055.0                    314
060.0                    309
065.0                    304
070.0                    299
075.0                    294
080.0                    289
085.0                    284
090.0                    279
095.0                    274
100.0                    269
105.0                    264
110.0                -655360
120.0                -655360
140.0                -655360
160.0                -655360
180.0                -655360
200.0                -655360
SENSOR_PREFERENCE_WEIGHT
0
500
400
300
200
100
0
0
0
0
```

```
PA_WEIGHT
883
SELECTED_WEIGHT
1000
NOT_SELECTED_WEIGHT
0
CURRENTLY_TASKED_WEIGHT
250
NOT_CURRENTLY_TASKED_WEIGHT
0
ABOVE_LINE_WEIGHT
250
BELOW_LINE_WEIGHT
0
REQUIRED_NO_PASSES_WEIGHT
82
ADDITIONAL_NO_PASSES_WEIGHT
50
MAX_ADDITIONAL_NO_PASSES_WEIGHT
50
```

Run #2

TEMPERATURE_RATIO_SCALE = 1e-06

Results:

```
number_generated = 363,  *number_accepted = 250
best...→cost = -0.8243348
best_generated_state →parameter[0] = 525
best_generated_state →parameter[1] = 165
best_generated_state →parameter[2] = 283
best_generated_state →parameter[3] =   59
best_generated_state →parameter[4] = 151

COST_REPEATING exit_status = 3
*number_accepted at best_generated_state →cost = 12
*number_generated at best_generated_state →cost = 20
```

Run #2 Resulting *weights.txt* File

```
ORBIT_DISTRIBUTION_WEIGHT
5.0
RANK_WEIGHTS
0
163
161
159
157
155
153
151
149
147
145
143
LOADING_WEIGHTS
005.0                 278
010.0                 273
015.0                 268
020.0                 263
025.0                 258
030.0                 253
035.0                 248
040.0                 243
045.0                 238
050.0                 233
055.0                 228
060.0                 223
065.0                 218
070.0                 213
075.0                 208
080.0                 203
085.0                 198
090.0                 193
095.0                 188
100.0                 183
105.0                 178
110.0             -655360
120.0             -655360
140.0             -655360
160.0             -655360
180.0             -655360
200.0             -655360
SENSOR_PREFERENCE_WEIGHT
0
500
400
300
200
100
0
0
0
0
```

```
PA_WEIGHT
525
SELECTED_WEIGHT
1000
NOT_SELECTED_WEIGHT
0
CURRENTLY_TASKED_WEIGHT
250
NOT_CURRENTLY_TASKED_WEIGHT
0
ABOVE_LINE_WEIGHT
250
BELOW_LINE_WEIGHT
0
REQUIRED_NO_PASSES_WEIGHT
59
ADDITIONAL_NO_PASSES_WEIGHT
151
MAX_ADDITIONAL_NO_PASSES_WEIGHT
151
```

Run #3

TEMPERATURE_RATIO_SCALE = 1e-06

Results:

```
number_generated = 557,  *number_accepted = 500
best...→cost = -0.835069
best_generated_state →parameter[0] =  10
best_generated_state →parameter[1] =  22
best_generated_state →parameter[2] = 105
best_generated_state →parameter[3] =  20
best_generated_state →parameter[4] =   1

COST_REPEATING exit_status = 3
*number_accepted at best_generated_state →cost = 0
*number_generated at best_generated_state →cost = 0
```

Resulting *weights.txt* file is the same as the default *weights.txt* file in Appendix A.

Run #4

TEMPERATURE_RATIO_SCALE = 1e-05

Results:

```
number_generated = 381,  *number_accepted = 250
best...→cost = -0.1677507
best_generated_state →parameter[0] = 993
best_generated_state →parameter[1] = 729
best_generated_state →parameter[2] = 993
best_generated_state →parameter[3] =   98
best_generated_state →parameter[4] = 257

COST_REPEATING exit_status = 3
*number_accepted at best_generated_state →cost = 33
*number_generated at best_generated_state →cost = 45
```

Run #4 Resulting *weights.txt* File

```
ORBIT_DISTRIBUTION_WEIGHT
5.0
RANK_WEIGHTS
0
727
725
723
721
719
717
715
713
711
709
707
LOADING_WEIGHTS
005.0                    928
010.0                    923
015.0                    918
020.0                    913
025.0                    908
030.0                    903
035.0                    898
040.0                    893
045.0                    888
050.0                    883
055.0                    878
060.0                    873
065.0                    868
070.0                    863
075.0                    858
080.0                    853
085.0                    848
090.0                    843
095.0                    838
100.0                    833
105.0                    828
110.0                -655360
120.0                -655360
140.0                -655360
160.0                -655360
180.0                -655360
200.0                -655360
SENSOR_PREFERENCE_WEIGHT
0
500
400
300
200
100
0
0
0
0
```

```
PA_WEIGHT
993
SELECTED_WEIGHT
1000
NOT_SELECTED_WEIGHT
0
CURRENTLY_TASKED_WEIGHT
250
NOT_CURRENTLY_TASKED_WEIGHT
0
ABOVE_LINE_WEIGHT
250
BELOW_LINE_WEIGHT
0
REQUIRED_NO_PASSES_WEIGHT
98
ADDITIONAL_NO_PASSES_WEIGHT
257
MAX_ADDITIONAL_NO_PASSES_WEIGHT
257
```

Bibliography

1. Bazaraa, Mokhtar S., John J. Jarvis, and Hanif D. Sherali. *Linear Programming and Network Flows* (Second Edition). New York: John Wiley and Sons, 1990.

2. Cerny, V. "Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm," *Journal of Optimization Theory and Applications,* 45:41-51, January 1985.

3. Chan, Yupo. Professor of Operations Research. Class notes, OPER 621, Multiple Criteria Decision Making. Graduate School of Engineering, Air Force Institute of Technology, Wright-Patterson AS OH, Fall Quarter 1994.

4. Chan, Yupo. Professor of Operations Research. Graduate School of Engineering, Air Force Institute of Technology, Wright-Patterson AS OH. Personal Interview, 13 October 1994.

5. Cherry, Paul R. "Sensor Tasking by the Space Defense Operations Center," *Proceedings of the 11th Space Surveillance Workshop.* 93 - 98. Lexington MA: Massachusetts Institute of Technology Lexington Lincoln Laboratory, 1 April 1993 (AD-A265120).

6. Collins, N. E., R. W. Eglese and B. L. Golden. "Simulated Annealing -- An Annotated Bibliography," *American Journal of Mathematical and Management Sciences*, 8:209-233, 1988.

7. Combs, Daniel E., Chief of Tasking, 1st Command and Control Squadron (AFSPC), Cheyenne Mountain AS CO. Personal Correspondence. 21 March 1994.

8. Combs, Daniel E., Chief of Tasking, 1st Command and Control Squadron (AFSPC), Cheyenne Mountain AS CO. Personal Interview. 21 May 1994.

9. Combs, Daniel E., Chief of Tasking. "Sensor Tasking," Briefing Slides. 1st Command and Control Squadron (AFSPC), Cheyenne Mountain AS CO.

10. Combs, Daniel E., Chief of Tasking, 1st Command and Control Squadron (AFSPC), Cheyenne Mountain AS CO. Telephone Interviews. 3 February 1994 - 27 September 1994.

11. Cooke, David G. "Space Surveillance -- The SMART Catalog," *AAS/AIAA Astrodynamics Conference.* 569-575. San Diego: Univelt, Inc., 1988.

12. Department of the Air Force. *Space Handbook.* AU-18. Maxwell AFB AL: Air University Press, December 1993.

13. De Vere, G. T. "Space Surveillance Network Sensor Contribution Analysis," *Proceedings of the 11th Space Surveillance Workshop.* 33 - 38. Lexington MA: Massachusetts Institute of Technology Lexington Lincoln Laboratory, 1 April 1993 (AD-A265120).

14. First Command and Control Squadron. Operations Center Job Aid 42, Tasking Groups. Cheyenne Mountain AS CO, 24 March 1994.

15. Goicoechea, Ambrose, Don R. Hansen, and Lucien Duckstein. *Multiobjective Decision Analysis With Engineering and Business Applications.* New York: John Wiley and Sons, 1982.

16. Ingber, Lester. "Simulated Annealing: Practice Versus Theory," *Mathematical Computer Modeling,* 18:29-58, 1993.

17. Ingber, Lester. *Adaptive Simulated Annealing (ASA) v3.17, README.ps.* Computer Software. GNU General Public License (GPL), ftp.caltech.edu pub/ingber directory, 28 June 1994.

18. Jackson, P. A. "Space Surveillance Satellite Catalog Maintenance," *AIAA/NASA/DOD Orbital Debris Conference: Technical Issues and Future Directions.* Paper No. 90-1339. Baltimore: American Institute of Aeronautics and Astronautics, April 1990.

19. Kirkpatrick, S., C. D. Gelatt and M. P. Vecchi. "Optimization by Simulated Annealing," *Science,* 220:671-680, 13 May 1983.

20. McEachin, Richard C. *An Investigation of Simulated Annealing Applied to Structural Optimization Problems.* MS Thesis, AFIT/GST/ENS/94M-08. Graduate School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1994.

21. Miller, J. G., Lead Engineer. "Sensor Tasking Prototype," Briefing Slides. Colorado Springs CO: The MITRE Corporation, 3 November 1993.

22. Miller, J. G., Lead Engineer, The MITRE Corporation, Colorado Springs CO. Telephone Interviews. 13 July 1994 - 1 October 1994.

23. Otten, R. and L. van Ginneken. *The Annealing Algorithm.* Norwell, MA: Kluwer Academic Publishers, 1989.

24. Reklaitis, G. U., A. Ravindran, and K. M. Ragsdell. *Engineering Optimization, Methods and Applications.* New York: John Wiley & Sons, Inc., 1983.

25. SPADOC 4C Sensor Tasking Prototype. Computer Software. The MITRE Corporation. Colorado Springs CO, 1994.

26. Toro, Radames, Ramon A. Barrios, and others. *Space Operations Orientation Course Handbook* (Third Edition). Colorado Springs CO: 21st Crew Training Squadron (AFSPC) and FlightSafety Services Corporation, August 1993.

27. Tovey, Craig A. "Simulated Simulated Annealing," *American Journal of Mathematical and Management Sciences,* 8:389-407, 1988.

28. van Laarhoven, P. and E. Aarts. *Simulated Annealing: Theory and Applications.* Eindhoven, The Netherlands: D. Reidel Publishing Company, 1987.

VITA

Captain Beth L. Petrick was born on 5 September 1967 in Pittsburgh, Pennsylvania. She grew up in a suburb of Pittsburgh and in 1985 graduated from Norwin High School. She then entered the Reserve Officers Training Corps at Carnegie-Mellon University in Pittsburgh. There she received a Bachelor of Science degree in Mathematics in 1989. After commissioning, her first assignment was to the 2nd Space Operations Squadron (2 SOPS) at Falcon AFB, Colorado. There, she performed crew duty for the Navstar Global Positioning System as a Satellite Operations Officer. After a year, she was trained as a Crew Commander, and stayed on crew duty for another six months. She was then placed in the Standardization and Evaluation Branch of the 2 SOPS. After a year there, she moved to the 50th Operations Group Standardization and Evaluation Division. It was from this assignment that she entered the Graduate School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio. Her follow-on assignment is to the 553rd Training Squadron, formerly the 21st Crew Training Squadron, at Peterson AFB, Colorado. At this assignment, she will be in charge of the orbital mechanics training for personnel entering the Space Control Center at Cheyenne Mountain AS, Colorado.

Permanent Address: 11297 Drop Rd.
North Huntingdon, PA 15642

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 1994	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE WEIGHTING SCHEME FOR THE SPACE SURVEILLANCE NETWORK AUTOMATED TASKER	**5. FUNDING NUMBERS**
6. AUTHOR(S) Beth L. Petrick, Capt, USAF	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT/GSO/ENS/94D-13
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) 1 CACS/CC 1 Norad Rd. Suite 3105 CMAS CO 80914-6009	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT Distribution Unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(Maximum 200 words)*

The purpose of this research is to find the best weighting scheme in the SPADOC 4C Sensor Tasking Prototype. This software, known as the Prototype Tasker, assigns a tasking to each sensor of the Space Surveillance Network. A tasking is a list of objects in which the sensor needs to gather positional data, called observations. One of the inputs to the Prototype Tasker is a user-definable weighting scheme. The goal is to find the weighting scheme that produces the most efficient taskings. This problem was solved using a Simulated Annealing algorithm. The values in the weighting scheme were the variables of the problem. The objective function was a measure of the goodness of each tasking produced. The Simulated Annealing algorithm was set to vary the weighting scheme and find the one that produces the tasking with the highest objective function. There were four successful executions of the simulated annealing algorithm. Each one produced a different weighting scheme. However, there were noticeable trends in the relative magnitudes of the weights. Also, it was noticed that a slight decrease in the number of observations on the taskings will increase the expected amount of information gathered by the Space Surveillance Network.

14. SUBJECT TERMS Simulated Annealing, Space Surveillance, Non-linear Programming, Zero-order Search, Probabilistic Search	**15. NUMBER OF PAGES** 93
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18. SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	**19. SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

★U.S.GPO:1990-0-273-271